Welcome to itucsdb1909's documentation!

Team: Anthology
Members:
- Mehmet Gencay Ertürk

Keep your favorite poems and books in Anthology keeps each users favorites and also community favorites. Each user can contribute to community lists, as in Wikipedia. Then user can pick their favorites in that list to shape his/her own list.

There are a different community lists for poems and books. Users can add, update or delete elements in community lists and changes are reflected to the user lists. User can also shape his/her own list by adding or deleting items from his/her lists.

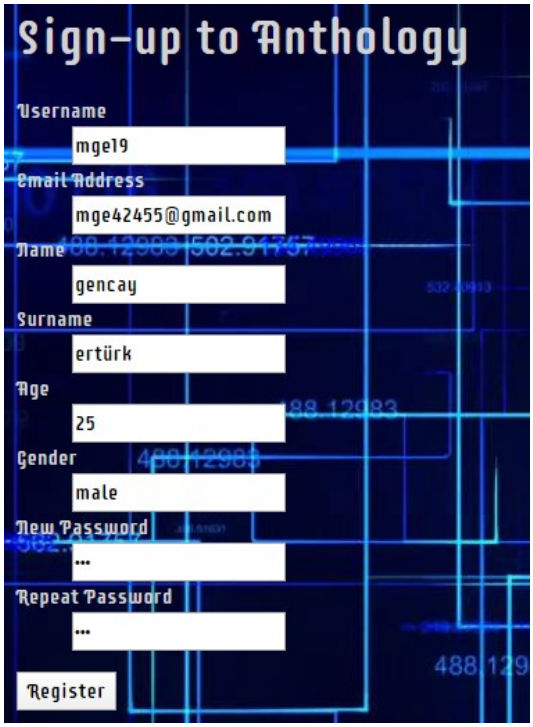
Contents:

User Guide

As a visitor of Anthology, you are welcomed with the homepage.

Authentication

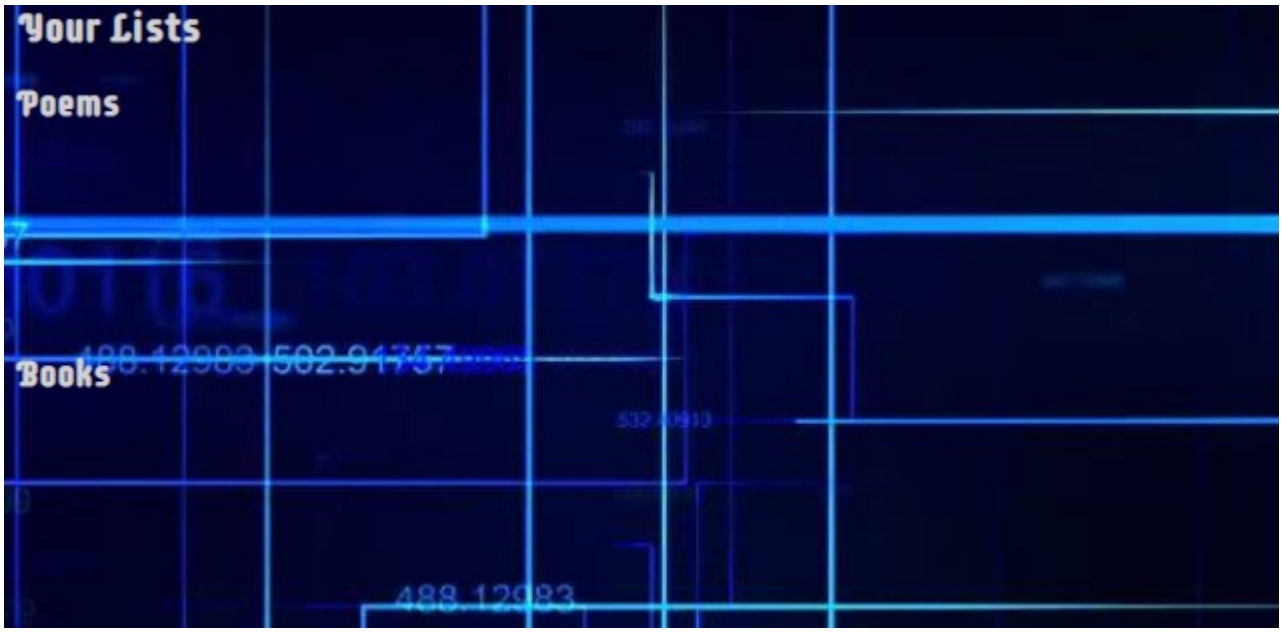
You should sign up to get full benefit of Anthology. For that, you should visit the sign up page.



With a successful sign-up you are redirected to the sign-in page.

Poems & User Lists

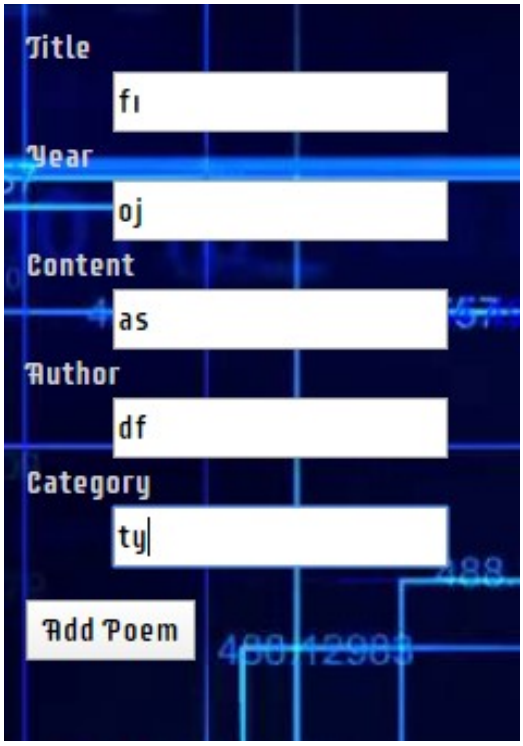
After the sign-in now you can start using Anthology. Initially, your lists are empty as it is shown below.



Now it’s time to visit the community poems/books. Let’s start with poems. Don’t forget that the very similar operations are applied for books.

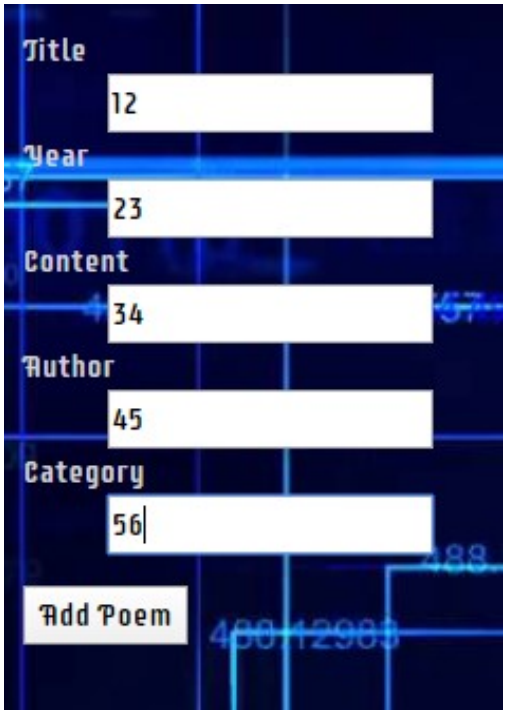
As an Anthology user, you are more than free to contribute the community poems. Here you can add, update or delete the poems you wish.

Let’s start by adding a poem.



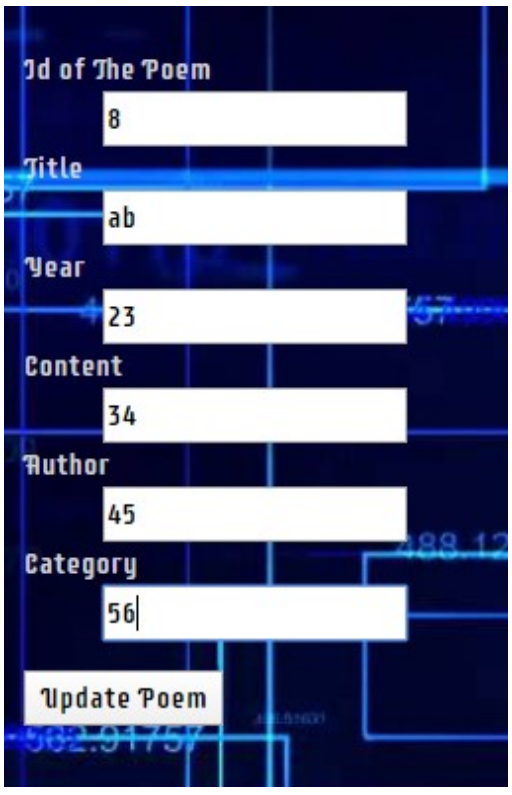
And here it is in the Community Poems,

We can add more poems,

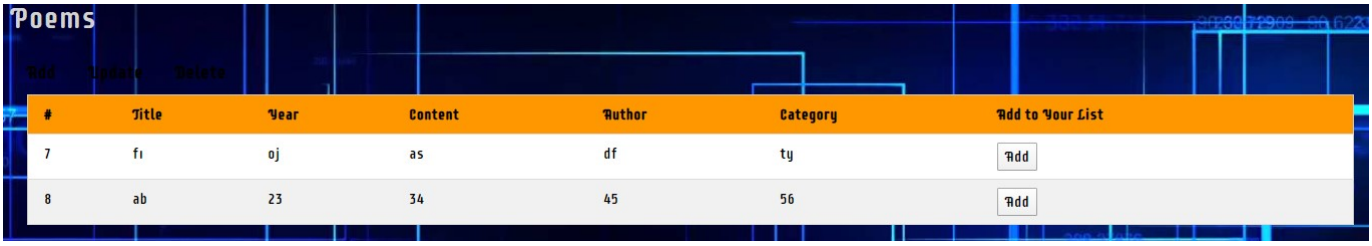


Our community list is growing,

We can update Community Poems,



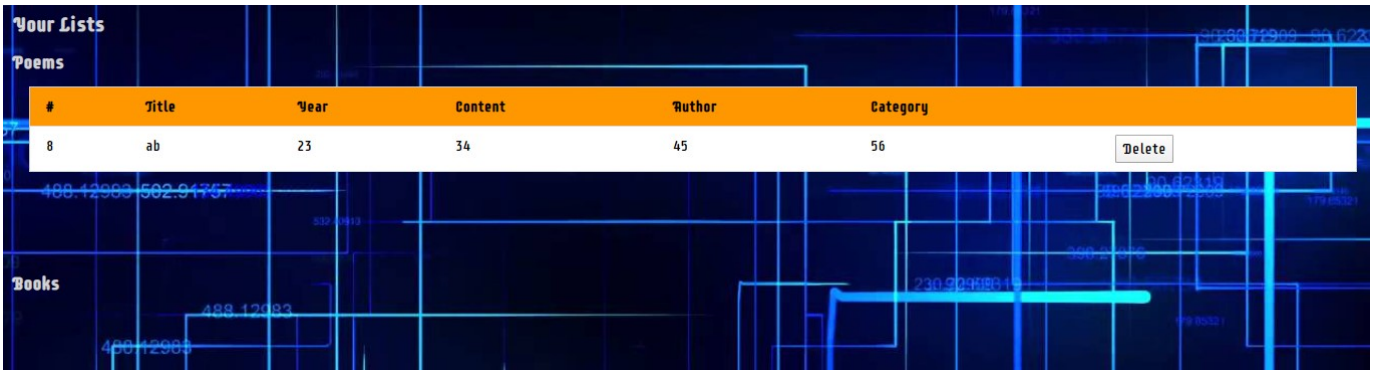
Here is the result,



We can delete poems too,

Here is our updated list,

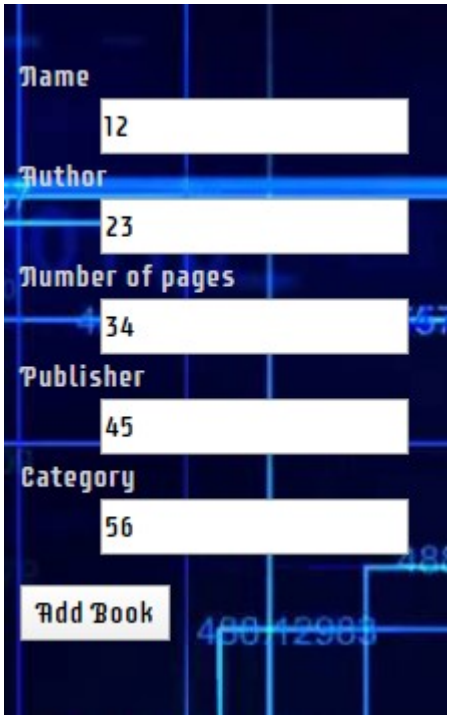
And here is our list;

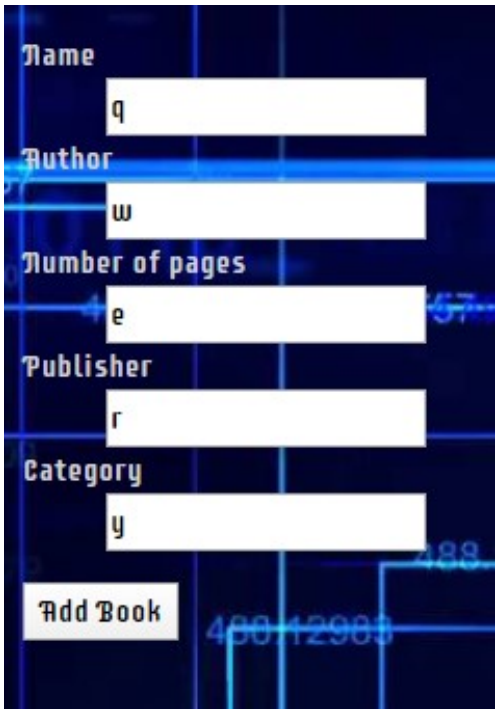


That’s all for the usage of Anthology. If you want more information, you can find similar operations done in books list from now on, Also, you can delete your favorites by clicking the Delete button. In addition, you can delete from community lists which’ll be cascaded to the user lists. These’ll be shown in the next section.

Books

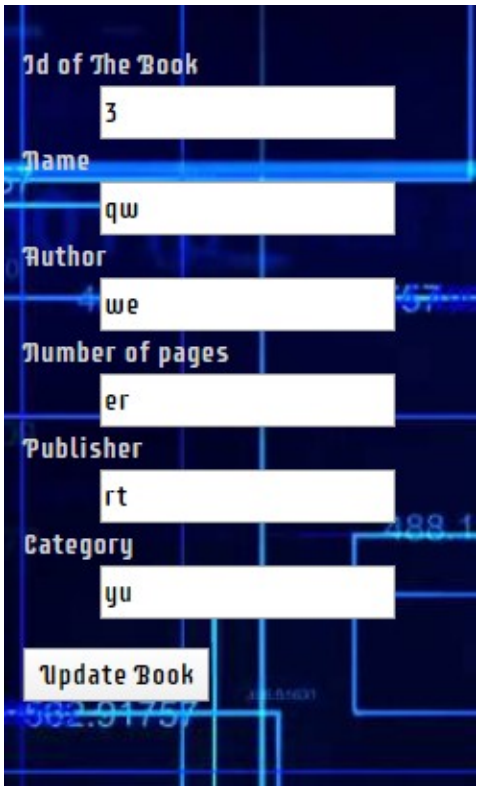
Let’s add couple of book;





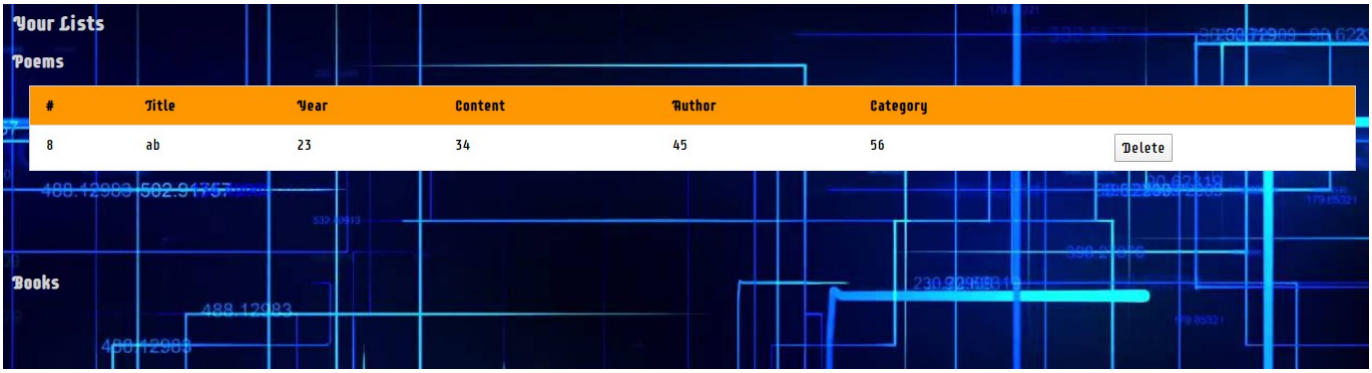
Here we go, this is the Community Books now:

We can also update/delete books from the list:

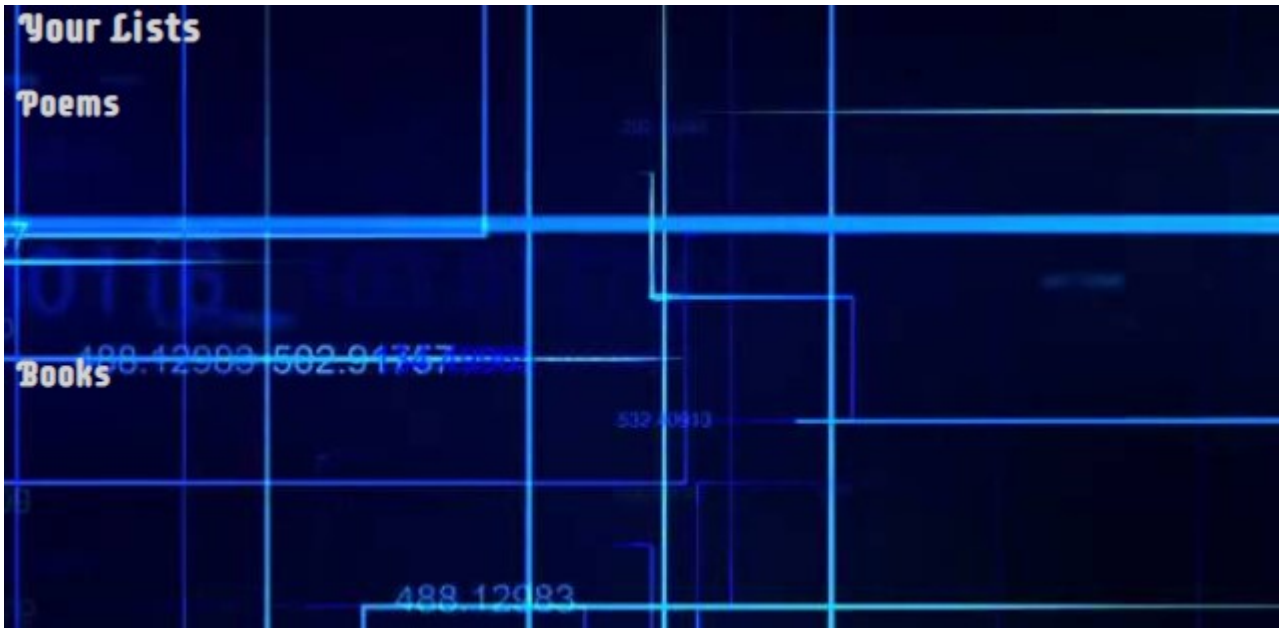


We can “Add” a book to our lists;

We can even “Delete” books from our lists. Let’s do it in “qw”:



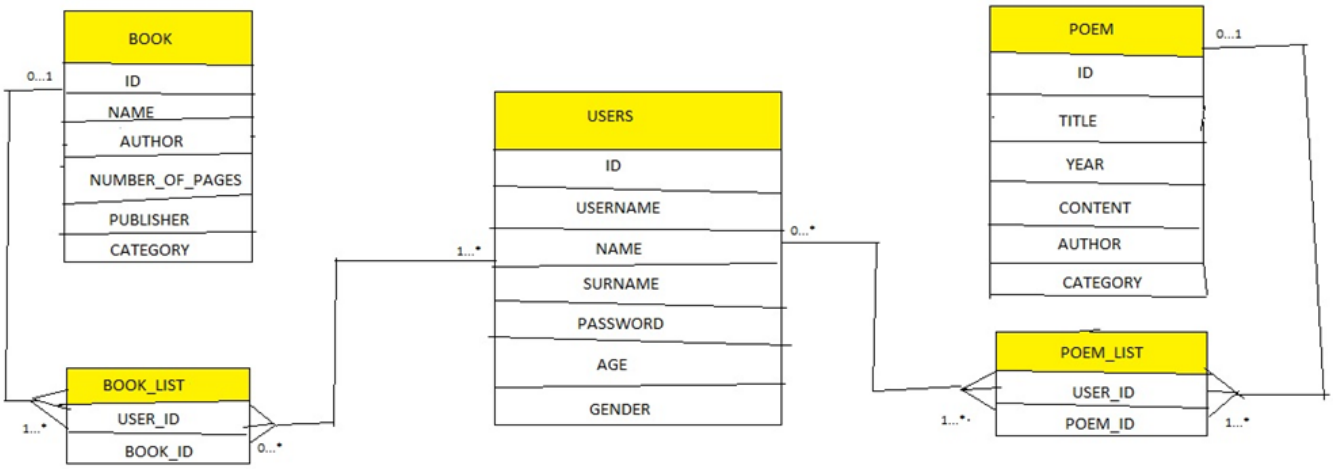
What happens to user lists when we delete an existing item (poem with ID 8) from the community lists? Here we go. As a reminder, here is the last situation of our list;



Developer Guide

Database Design

There are three main tables; **USERS, POEM** and **BOOK**. USERS table is designed to keep the information for user. poem table is for poems, and book table is for books. There are also two additional tables; **POEM_LIST** and **BOOK_LIST**. These are designed to keep user's favorited books/poems. These keep ID's for USER and POEM/BOOK.



Code

The code can be explained in three parts.

- Python & SQL

- HTML/CSS

Python & SQL

There are the following python(.py) files;

- server.py
- dbinit.py
- forms.py
- db_table_operations.py
- config.py

server.py

This is the main python file for the project. It renders pages, initializes and transfers variables and realizes methods with necessary HTML requests.

Firstly, the following packages are imported.

```
from flask import Flask, render_template, request, redirect
from flask_bootstrap import Bootstrap
from flask_login import login_user,logout_user,UserMixin,
LoginManager

from forms import *
from config import Config
from db_table_operations import *
from werkzeug.security import generate_password_hash
```

Then configuration, bootstrap and LoginManager is initialized. LoginManager is used for authentication.

```
  app = Flask(__name__)
  app.config.from_object(Config)
  Bootstrap(app)

  login = LoginManager(app)
```

The rest of the server.py works for routes. For example, this renders homepage.html when main route (“/”) is accessed.

```
@app.route("/")
def home_page():
```

```
   return render_template("homepage.html")
```

For the sign-up operation, route and necessary HTML requests are announced. Then a form is created (forms will be discussed in the forms.py. When user clicks to the “Register” button, then a new user object is created with information provided by the user. Then the user object is sent to insert_user which will work to insert user to the USERS table. With a successful sign-up operation, users are redirected to the signin page.

```
@app.route('/signup', methods=['GET', 'POST'])
def signup_page():
    form = RegistrationForm(request.form)
    if request.method == 'POST' and form.validate():
     user = User(form.username.data, form.name.data, form.surname.data,
                    form.password.data, form.age.data, form.gender.data)
      insert_user(user)
       return redirect('/signin')
```

The similar operations are realized for sign in; route, HTML requests and form. When user clicks “Sign in” button, find_user_by_username method runs a SQL query to find the username in the database. If it can, Flask Login app is informed about the signed up user. The username is also shown in the menu bar at Mylists page.There is also a sign out route.It signs the user out and redirects to the homepage.

```
@app.route('/signin', methods=['GET', 'POST'])
def signin_page():

form = LoginForm()
if form.validate_on_submit():

found_user = find_user_by_username(form.username.data)
  user = User(found_user[1], found_user[2], found_user[3], foun

          found_user[5], found_user[6], found_user[7])
   user.set_id(found_user[0]) # to load user id
if user is None or not check_password(user.password, form.password.data):

  print("User signing failed")
 return redirect('/signin')

  login_user(user, remember=form.remember_me.data)
 print("User signed successfully")

     return redirect('/mylists')
 return render_template("signin.html", form=form)

@app.route('/signout')
def signout_page():
    logout_user()
    return redirect('/')
```

From this point on, there are the following methods left in the server.py;

- mylists
- poems (add/update/delete)
- books (add/update/delete)

Operations for poems and books are very similar. For the sake of simplicity, mylists and poems will be discussed.

Mylists page tries to get elements from user's own lists as a first step. If there is any element exists, it shows them.

Also, mylists page has "Delete" button for items in the user's list. When it is clicked, it takes the related elements id (book or poem and user's id. Then it sends it to delete_poem (delete_book for book) operation to run SQL query to delete it from the database.

As it can be noticed, mylists page provides user_poems and user_books variables while it is rendering the html file. These variables are going to be processed with the html file to fill the tables.

```
@app.route('/mylists', methods=['GET', 'POST'])
def mylists_page():
   user_poems   =   userlist_get_poems()
   user_books = userlist_get_books()

    if request.method == 'POST':
        poem_id = request.form['poem_id']
        user_id = request.form['user_id']
        book_id = request.form['book_id']

        if poem_id != "0":  # request to delete poem
            userlist_delete_poem(user_id, poem_id)
           print("delete poem with id ="poem_id)
           returnn redirect('/mylists')

      elif book_id != "0":  # request to delete book
            userlist_delete_book(user_id, book_id)
           returnn redirect('/mylists')

   return render_template("mylists.html",
                          user_poems=user_poems,
                          user_books=user_books)
```

Poems page works similarly with the mylists page. It shows the poems in the poem table in sorted order. It has "Add" button for user to add the selected poem to his/her lists. Added elements are going to be shown in users list in

mylists page.

```
def poems_page():
    poems = get_poems()

    if request.method == 'POST':
        poem_id = request.form['poem_id']
       user_id = request.form['user_id']
       userlist_add_poem(user_id, poem_id)
       return redirect('/mylists')

     return render_template("poems.html", poems=sorted(poems))
```

Adding a poem is fairly simple and similar to the user sign up operation. It works with PoemAddForm and creates a poem object with provided information. Then to run SQL query, it sends the poem object to insert_poem() method.

```
@app.route('/mylists/poems/add', methods=['GET', 'POST'])
def poems_add_page():
    form = PoemAddForm(request.form)
    if request.method == 'POST' and form.validate():
       poem = Poem(form.title.data, form.year.data, form.content.data,
                   form.author.data, form.category.data)
        insert_poem(poem)
        return redirect('/mylists/poems')
      return render_template("add_poems.html", form=form)
```

Poem update page also gets id to be deleted.

```
def poems_update_page():
     form = PoemUpdateForm(request.form)
      if request.method == 'POST' and form.validate():
       poem = Poem(form.title.data, form.year.data, form.content.data,
                   form.author.data, form.category.data)
         poem_id = form.id.data  #    #which poem to update
        update_poem_id, poem)
         return redirect('/mylists/poems')

   return render_template("update_poems.html", form=form)
```

Poem delete page gets id that will be deleted from the database table, poem.

```
def poems_delete_page():
   form = PoemDeleteForm(request.form)
```

```
    id = form.poem_id.data
    if id.__len__() > 0:
      delete_poem(id)
      return redirect('/mylists/poems')
    return render_template("delete_poems.html", form=form)
```

server.py also includes User, Poem and Book classes and the login manager in the User class.

```
class User(UserMixin):
login = LoginManager(app)

 def __init__(self, username, name, surname, email, password, age

 self.id = 0

  self.username = username
self.name = name
 self.surname = surname
self.email = email
 self.password = password
self.age = int(age)
 self.gender = gender
print("User object created")


  @login.user_loader
def load_user(id):

 if id == 0:

     print("User not logged in, id is ", id)
return

 else:


   found_user = find_user_by_id(int(id))
  return found_user

    def __repr__(self):
      return '<User %r>' % self.username

    def set_id(self, id):
      self.id = id

    def get_id(self):
```

```
        return self.id;

class Book:
   def __init__(self, name, author, number_of_pages, publisher, category):
         self.name= name
         self.author = author
         self.dnumber_of_pages = number_of_pages
        self.publisher = publisher
          self.category = category
          print("Book object created")


class Poem:
   def __init__(self, title, year, content, author, category):
           self.title = title
           self.year = year
           self.content = content
           self.author = author
           self.category = category
      print("Poem object created")
```

dbinit.py:dbinit.py is used for database initialization. It runs INIT_STATEMENTS which has the purpose to create the tables, USERS, POEM, BOOK.

```
INIT_STATEMENTS = [

    "DROP TABLE IF EXISTS USERS CASCADE", # to test changes
   quickly  "DROP TABLE IF EXISTS BOOK CASCADE",
    "DROP TABLE IF EXISTS POEM CASCADE",
    "DROP TABLE IF EXISTS BOOK_LIST CASCADE",
   "DROP TABLE IF EXISTS POEM_LIST CASCADE",

    "CREATE TABLE IF NOT EXISTS USERS("
        "ID SERIAL,"
        "USERNAME VARCHAR(30) NOT NULL,"
       "NAME VARCHAR(30),"
        "SURNAME VARCHAR(30),"
        "EMAIL VARCHAR(30),"
        "PASSWORD VARCHAR(100),"
        "AGE VARCHAR(8),"
        "GENDER VARCHAR(15),"
        "PRIMARY KEY(ID)"
    ")",

    "CREATE TABLE IF NOT EXISTS BOOK("
   "ID SERIAL,"
    "NAME VARCHAR(30),"
   "AUTHOR VARCHAR(30),"
```

```
    "NUMBER_OF_PAGES VARCHAR(8),"
   "PUBLISHER VARCHAR(30),"
    "CATEGORY VARCHAR(8),"
   "PRIMARY KEY(ID)"  ")",


    "CREATE TABLE IF NOT EXISTS POEM("
   "ID SERIAL,"
    "TITLE VARCHAR(40),"
    "YEAR VARCHAR(8),"
    "CONTENT VARCHAR(8),"
   "AUTHOR VARCHAR(30),"
    "CATEGORY VARCHAR(30),"
   "PRIMARY KEY(ID)"  ")",
```

Notice that USER_ID and BOOK_ID/POEM_ID are the primary keys for these tables. Also, they got cascading.

```
    "CREATE TABLE IF NOT EXISTS BOOK_LIST("
    "USER_ID INTEGER REFERENCES USERS(id) ON DELETE
   CASCADE,"  "BOOK_ID INTEGER REFERENCES BOOK(id) ON DELETE
   CASCADE,"  "PRIMARY KEY(USER_ID, BOOK_ID)"
    ")",

    "CREATE TABLE IF NOT EXISTS POEM_LIST("
    "USER_ID INTEGER REFERENCES USERS(id) ON DELETE
   CASCADE,"  "POEM_ID INTEGER REFERENCES POEM(id) ON DELETE
   CASCADE,"  "PRIMARY KEY(USER_ID, POEM_ID)"
    ")",
]
```

forms.py:forms.py is used to create forms necessary for the related operations.

```
class LoginForm(FlaskForm):
    username = StringField('Username', validators=[DataRequired()])
   password = PasswordField('Password', validators=[DataRequired()])
   remember_me = BooleanField('Remember Me')
    submit = SubmitField('Sign In')


class RegistrationForm(Form):
    username = StringField('Username', [validators.Length(min=1,
   max=50)])  email = StringField('Email Address',
   [validators.Length(min=0, max=50)])  name = StringField('Name',
   [validators.Length(min=0, max=50)])  surname =
   StringField('Surname', [validators.Length(min=0, max=50)])  age
   = StringField('Age', [validators.Length(min=0, max=3)])  gender
   = StringField('Gender', [validators.Length(min=0, max=5)])
```

```
    password = PasswordField('New Password', [
         validators.DataRequired(),
         validators.EqualTo('confirm', message='Passwords must match'
    ])
     confirm = PasswordField('Repeat Password')

# POEM OPERATIONS
class PoemAddForm(Form):
     title = StringField('Title', [validators.Length(min=1, max=4)])
    year = StringField('Year', [validators.Length(min=0, max=50)])
    content = StringField('Content', [validators.Length(min=0,
    max=50)])  author = StringField('Author',
    [validators.Length(min=0, max=50)]  category =
    StringField('Category', [validators.Length(min=0, max=50)])
class PoemUpdateForm(Form):
 id = StringField('Id of The Poem', [validators.Length(min=1,
max=4)])  title = StringField('Title', [validators.Length(min=0,
max=50)])  year = StringField('Year', [validators.Length(min=0,
max=50)])  content = StringField('Content, [validators.Length(min=0,
max=50)])  author = StringField('Author', [validators.Length(min=0,
max=50)])  category = StringField('Category',
[validators.Length(min=0, max=50)])
class PoemDeleteForm(Form):
     poem_id = StringField('Id', [validators.Length(min=1, max=5)])

#BOOK OPERATIONS
class BookAddForm(Form):
     name = StringField('Name', [validators.Length(min=1, max=4)])
    category = StringField('Category', [validators.Length(min=0,
    max=50)])  number_of_pages = StringField('Number of Pages',
    [validators.Length(min=0, max=50)])  publisher=
    StringField('Publishher', [validators.Length(min=0, max=50)]  author
    = StringField('Author', [validators.Length(min=0, max=50)])
class BookUpdateForm(Form):
    id = StringField('Id of The book', [validators.Length(min=1, max=4)])
  name = StringField('Name', [validators.Length(min=0, max=50)])  category =
  StringField('Category', [validators.Length(min=0, max=50)])
  number_of_pages = StringField('Number of Pages', [validators.Length(min=0,
  max=50)])  publisher= StringField('Publishher', [validators.Length(min=0,
  max=50)]  author = StringField('Author', [validators.Length(min=0,
  max=50)])

class BookDeleteForm(Form):
    book_id = StringField('Id', [validators.Length(min=1, max=4)])
```

db_table_operations.py: This file is to execute SQL queries needed for database operations.

Insert a user;

```
# USER AUTHENTICATION TABLE OPERATIONS #
def insert_user(object):
    query ='INSERT INTO USERS (USERNAME, NAME, SURNAME, EMAIL,
    PASSWORD,AGE,GENDER)' /
         'VALUES(%s, %s, %s, %s, %s, %s, %s)'
      url = get_db_url()
      with dbapi2.connect(url) as connection:
         cursor = connection.cursor()
           print("User pw:" + object.password)


        cursor.execute(query, (object.username, object.name, object.
surname,object.email,object.password, object.age, object.gender)
cursor.close()
```

object is user object which has fields like username, name, password etc. They are executed as strings(%s). Also notice that the password is hashed for security.

Find a user; Columns are selected with SELECT operation.

```
def find_user_by_username(username):
    query = "SELECT * FROM USERS WHERE USERNAME = %s"
   url = get_db_url()
    with dbapi2.connect(url) as connection:
        cursor = connection.cursor()
        rows_count = cursor.execute(query,(username,))
       print("User is found in DB")

        found_user = cursor.fetchone()
        cursor.close()
        return found_user


def find_user_by_id(id):
    query = "SELECT * FROM USERS WHERE ID = %s"
   url = get_db_url()
    with dbapi2.connect(url) as connection:
        cursor = connection.cursor()
        rows_count = cursor.execute(query,(id,))
       print("User is found in DB")
        found_user = cursor.fetchone()
        cursor.close()
        return found_user
```

Check password; User input and user password are compared.

```
def check_password(user_password_hash, form_password):
    # compare the passwords

   return user.password==form.password
```

Add/Update/Delete elements from User Lists (i.e. mylists);

```
# USER LIST OPERATIONS
def userlist_add_book(user_id, book_id):


    query = 'INSERT INTO BOOK_LIST (USER_ID, book_ID) VALUES(%s,
   %s)'  url = get_db_url()
    with dbapi2.connect(url) as connection:
        cursor = connection.cursor()
        cursor.execute(query, (user_id, book_id))
       cursor.close()

def userlist_add_poem(user_id, poem_id):
    query = 'INSERT INTO POEM_LIST (USER_ID, poem_ID) VALUES(%s,
   %s)'  url = get_db_url()
    with dbapi2.connect(url) as connection:
        cursor = connection.cursor()
        cursor.execute(query, (user_id, poem_id))
       cursor.close()
```

User's lists, i.e, POEM_LIST and BOOK_LIST keeps only the ID's of the poems/books and user. So it is necessary to use JOIN operation to find out the row related with these ID's.

```
  def userlist_get_poems():

   query ='SELECT POEM.* FROM POEM_LIST ' \
           'INNER JOIN POEM ON POEM_LIST.POEM_ID = POEM.ID '
   \    'INNER JOIN USERS ON POEM_LIST.USER_ID = USERS.ID'
   url = get_db_url()
    with dbapi2.connect(url) as connection:
        cursor = connection.cursor()
       cursor.execute(query)  poems
       = cursor.fetchall()
       cursor.close()
        return poems

def userlist_get_books():
   query ='SELECT BOOK.* FROM BOOK_LIST ' \
           'INNER JOIN BOOK ON BOOK_LIST.BOOK_ID = BOOK.ID '
   \    'INNER JOIN USERS ON BOOK_LIST.USER_ID = USERS.ID'
   url = get_db_url()
    with dbapi2.connect(url) as connection:
       cursor=connection.cursor()
       cursor.execute(query)
       books=cursor.fetchall()
       cursor.close()
```

```
        return books
```

Delete operation finds the element with provided ID and deletes it from the related table.

```
def userlist_delete_poem(user_id, poem_id):
    query = "DELETE FROM POEM_LIST WHERE USER_ID = CAST(%s AS INTEGER)  url =
   get_db_url()
    with dbapi2.connect(url) as connection:
         cursor = connection.cursor()
           cursor.execute(query, (user_id, poem_id))
           cursor.close()
         print("Poem with id " + poem_id + " from user with id "
        + user_id + "deleted")

def userlist_delete_book(user_id, book_id):
    query = "DELETE FROM BOOK_LIST WHERE USER_ID = CAST(%s AS
   INTEGE  url = get_db_url()
    with dbapi2.connect(url) as connection:

        cursor = connection.cursor()
          cursor.execute(query, (user_id, book_id))
          cursor.close()
          print("Book with id " + book_id + " from user with id "
         + "deleted")
```

Rest of the poem table operations are similar to the former operations.

```
# POEM TABLE OPERATIONS #
def get_poems():
     query ='SELECT * FROM POEM'
      url = get_db_url()
      with dbapi2.connect(url) as connection:
     cursor = connection.cursor()
     cursor.execute(query)  poems =
     cursor.fetchall()
     cursor.close()
     return poems

def insert_poem(poem):
    query ='INSERT INTO POEM(TITLE, YEAR, CONTENT, AUTHOR,
   CATEGORY) VALUES(%s, %s, %s, %s, %s)'  url = get_db_url()
    with dbapi2.connect(url) as connection:
        cursor = connection.cursor()
        cursor.execute(query, (poem.content ,poem.author,
       poem.category))
        cursor.close()
def update_poem(poem_id, poem):
   query = "UPDATE POEM " \
```

```
            "SET TITLE = %s, " \
            "YEAR = %s, " \
            "CONTENT = %s, " \
           "AUTHOR = %s, " \
              "CATEGORY = %s " \
            "WHERE ID = %s "
   url = get_db_url()
     with dbapi2.connect(url) as connection:
       cursor = connection.cursor()
       cursor.execute(query, (poem.title, poem.year,
      poem.content ,poem.author, poem.category, poem_id,)
      # id = cursor.fetchone()[0]  # get the inserted row's id
      cursor.close()
       print("poem with id " + poem_id + " deleted")
       # return id
```

Operations in BOOK table are pretty the same. Though, they are provided to provide details for SQL operations.

```
# BOOK TABLE OPERATIONS #
def get_books():

    query ='SELECT * FROM BOOK'
     url = get_db_url()
     with dbapi2.connect(url) as connection:
       cursor = connection.cursor()
      cursor.execute(query)
      books = cursor.fetchall()
      cursor.close()
      return books


   def insert_book(book):
   query ='INSERT INTO book (NAME, AUTHOR, NUMBER_OF_PAGES, PUBLISHER,
  CATEGORY) VALUES(%s, %s, %s, %s, %s)'  url = get_db_url()
    with dbapi2.connect(url) as connection:
         cursor = connection.cursor()
           cursor.execute(query, (book.name, book.author,
          book.number_of_pages,  book.publishher, book.category))
          cursor.close()


def update_book(book_id, book):
  query = "UPDATE BOOK " \
            "SET NAME = %s, " \
            "AUTHOR = %s, " \
            "NUMBER_OF_PAGES = %s, " \
           "PUBLISHER = %s, " \
           "CATEGORY = %s " \
```

```
                "WHERE ID = %s "
          url = get_db_url()
          with dbapi2.connect(url) as connection:
cursor = connection.cursor()
  cursor.execute(query, (book.namebook.name,
 book.author, book.number_of_pages, book.publisher, book.category, book_id,))
           cursor.close()
            print("book with id " + book_id + "

   deleted") def delete_book(book_id):
       query = "DELETE FROM BOOK WHERE ID = CAST(%s AS
      INTEGER)"  url = get_db_url()
       with dbapi2.connect(url) as connection:
            cursor = connection.cursor()
              cursor.execute(query, (book_id,))
              cursor.close()
              print("book with id " + book_id + "deleted")
```

Lastly, config.py has the secret key for security.

HTML/CSS

There are a differentiated HTML code for each page. Though, they all share the common one, layout.html. Let’s start with it.

Let’s start with CSS part of the layout.html.

```
.. code-block:: python

        <style>
     html, body, h1, h2, h3, h4, h5, h6
    {  font-family: 'Supermercado One',
    serif;  }
```

The code part above, font is determined. Then the rest of the code layouts the general style.

```
      {box-sizing: border-box;

  }

  /* Create two equal columns that floats next to each other
  */ .column {
    float: left;
   padding: 10px;
```

```
   padding-right: 15px;
  }

  .left {
   width: 85%;
  }

  .right {
   width: 15%
  }

  {# responsive to screen size #}
  @media screen and (max-width: 600px) {
   .column {
     width: 100%;
   }
  }

  /* Clear floats after the columns
  */ .row:after {
    content: "";
   display: table;
   clear: both;
  }
  </style>
```

Below you can see the jquery and Bootstrap scripts initializations.

```
<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquer
<script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/boots
```

Now we create “title” block which will be different for each page.

```
<title>{% block title %}{% endblock %}</title>
```

After, we create our menu. It is designed as two lines. The first one consists of couple links for necessary redirections. Because it’s same for all the pages, it is not labeled as block, as in the second menu bar. Second menu bar includes links to the community lists. It also has a welcome message with included username. Because it’s only necessary for mylists page it is labeled as secondmenubar.

```
<div class="w3-bar w3-text-yellow w3-large">
   <h5>
        <a href="/mylists"><button class="w3-bar-item w3-button w3-ri
       <a href="/signup"><button class="w3-bar-item w3-button w3-rig
       <a href="/signin"><button class="w3-bar-item w3-button w3-rig
```

```
        <a href="/"><button class="w3-bar-item w3-button w3-left"><b
   </h5>
</div>

{% block secondmenubar %}
<div class="w3-bar w3-text-brown w3-large w3-border-top w3-border-bot
   <h5>
        <a href="/mylists"><button class="w3-bar-item w3-button w3-le
        <a href="/mylists/poems"><button class="w3-bar-item w3-butto
        <a href="/mylists/books"><button class="w3-bar-item w3-butto
        <button class="w3-bar-item w3-button w3-right"><b>Welcome {{
    </h5>
</div>
{% endblock %}
```

Then create a block for body. It's empty for the homepage.

```
<body>
    {% block content %}{%  endblock %}
</body>
```

Create the footer;

```
{   % block footer %}
<footer class="w3-bottom w3-container w3-text-gray">
    <p>Developed by Mehmet Gencay Ertürk</p>
</footer>
{% endblock %}
```

Block for additional styles;

```
{% block additional_styles %}
<style>
  body {
    padding-left: 10px;
   }
</style>
{% endblock %}
```

That's it for layout.html. All the other pages are derived from this page. While it is common, they differ in contents in the blocks. There are 12 more html pages.

- **User Authentication:** signup.html, signin.html
- **Homepage:** homepage.html
- **User Lists:** mylists.html

- **Community Lists for Poems:** poems.html, add_poems.html, update_poems.html, delete_poems.html
- **Community Lists for Books:** books.html, add_books.html, update_books.html, delete_books.html

Almost all the pages are created to work with forms. The forms has fields according to the related database tables.

User Authentication *signup.html* The technical part of the page can be seen below:

```
{% block content %}
        <h1>Sign-up to Anthology</h1>
 <form method=post>
  <dl>
    {{ render_field(form.username) }}
   {{ render_field(form.email) }}
   {{ render_field(form.name) }}
    {{ render_field(form.surname) }}
   {{ render_field(form.age) }}
    {{ render_field(form.gender) }}
   {{ render_field(form.password) }}
   {{ render_field(form.confirm) }}
  </dl>
  <p><input type=submit
 value=Register>
{% endblock %}
```

Each form element is rendered in the html.

User Authentication *signin.html* In sign-in, username and password is asked. Also, there is an option to “Remember” the user later on.

```
{% block title %} Sign-in for Anthology {% endblock %}
{% block content %}
    <h1>Log In To Anthology</h1>
    <form action="" method="post" novalidate>
        {{ form.hidden_tag() }}
        <p>
            {{ form.username.label }}<br>
       {{ form.username(size=32) }}  </p>
        <p>
            {{ form.password.label }}<br>
       {{ form.password(size=32) }}  </p>
```

```
        <p>{{ form.remember_me() }} {{ form.remember_me.label }}</p>
    <p>{{ form.submit() }}</p>
     </form>
     {% if error %}
         <p class="error">ERROR: {{ error }}</p>
     {% endif %}

{% endblock %}
```

Homepage *homepage.html* Homepage has a difference in the additional styles here. A specific background image is implemented.

```
{% block additional_styles %}
     <meta name="viewport" content="width=device-width, initial-
    scale=  <style>
       body {
             background-image: url('/static/images/bg01.jpg');
            height: 100%;
             background-repeat: no-repeat;
             background-size: cover;
             position: relative;
       }
     </style>

{% endblock %}
```

At this point, the remaining pages are the following:

- **User Lists:** mylists.html

-Community Lists for Poems:poems.html, add_poems.html, update_poems.html, delete_poems.html

- Community Lists for Books:books.html, add_books.html, update_books.html, delete_books.html

Because the operation is very similar, to keep the documentation neat, only the technical parts will be discussed.

User Lists: mylists.html MyLists is the page shows user's own lists. It creates table and fills it with user_poems list. user_poems is provided to html in server.py.

```
{% if user_poems %}
    <table class="w3-table-all">
      <thead>
        <tr class="w3-orange">
           <th>#</th>
           <th>Title</th>
```

```
          <th>Year</th>
          <th>Content</th>
         <th>Author</th>
         <th>Category</th>
          <th></th>
                      </tr>
                     </thead>
        {% for poem in user_poems %}  <tr>
            class="w3-text-black">
        <td>{{  poem[0]  }}</td>
      <td>{{   poem[1]   }}</td>
      <td>{{   poem[2]   }}</td>
      <td>{{   poem[3]   }}</td>
      <td>{{   poem[4]   }}</td>
      <td>{{ poem[5] }}</td>

        <form method=post>

              {# get poem id#}

            <td>
                 <input type="hidden" name="poem_id" value="{{ mov
                <input type="hidden" name="user_id" value="{{ curr
                <input type="hidden" name="book_id" value="0">
                 <input type=submit value="Delete">
               </td>
             </form>

       </tr> {% endfor %}

    </table>
  </div>
     {% endif %}
```

The very same operation is realized for movies.html and musics.html.

Community Lists for Poems & Books :

Adding, updating and deleting poem and book operations are pretty similar. They process the form fields as text boxes, and with the click of the button, the information is transfered to the related variable and it's processed in the database.

```
<form method=post>
    <dl>
       {{ render_field(form.title) }}
      {{ render_field(form.author) }}
```

```
  {{ render_field(form.number_of_pages) }}
 {{ render_field(form.publisher) }}
  {{ render_field(form.category) }}
</dl>
  <p><input type=submit value="Add poem">
```

Parts Implemented by Member Name

All the parts are implemented by Mehmet Gencay Ertürk.