CS421/621- Advance Web Application Development

Week 7

-Flask / 2-

Professor : Mahmut Unan – UAB CS

Agenda

- Flask
  - Templates
  - Inheritance
  - url_form
  - Filters
  - Forms with Flask
  - Error Handler
  - SQL Databases with Flask
  - SQLite
  - SQLAlchemy
  - Migration

Final Project

Project - Final Submission:
August 6th 2021 Friday
11:59 pm

Idea Survey Feedback

Deliverables

1) Declaration of Independent Completion
   - Canvas/Files/misc/
   - Clearly explain who did what…

2) Project Files
   - All the required files
   - .zip format → Canvas
   - include readme.txt file for the instructions

Deliverables / 2

- 3) Project Report
  - Introduction (The purpose of the project)
  - Technologies (All the technologies you used for backend and front end, explain why you choose them)
  - Results (include as many screen shots as needed)
  - Discussions / Future Works
  - References

  * Make sure you follow the Academic Honesty and Honor Code

Deliverables / 3

- 4) Project Presentation
  - Present your work and record it
    - You can use zoom to record your screen
    - Explain each pages/features
  - Upload the video to the Canvas

Templates

- So far we have only returned back HTML elements manually through a python string
- Generating HTML content from Python code is not a good idea, especially when variable data and Python language elements like conditionals or loops need to be put.
- Realistically we will want to connect a view function to render HTML templates
- We will return the output of a function bound to a certain URL in the form of HTML.

https://www.udemy.com/python-and-flask-bootcamp-create-websites-using-flask/learn/lecture/9340994?start=0#questions

Flask Templates

- Flask will automatically look for HTML templates in the templates directory
- We can render templates simply by importing the **render_template** function from flask and returning an .html file from our view function

https://www.udemy.com/python-and-flask-bootcamp-create-websites-using-flask/learn/lecture/9340994?start=0#questions

Exercise 1

- Get into your flask environment
- Create a folder → **flask_example**
  - Create a **main.py** file under flask_example folder
- Create a subfolder → **templates**
  - Create a **main.html** file under templates folder

```html
<!DOCTYPE html>
<html lang="en" dir="ltr">
  <head>
    <meta charset="utf-8">
    <title></title>
  </head>
  <body>
    <h1>welcome to Flask exercise 1</h1>


  </body>
</html>
```

- We have our html file, now we will link it to our python file. Open main.py and put the following command

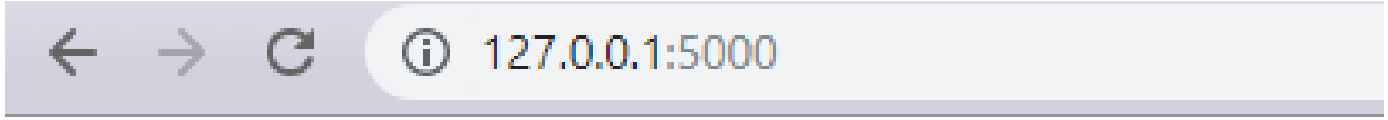
```python
from flask import Flask, render_template
app = Flask (__name__)

@app.route ('/')
def index ():
    return render_template('main.html')

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

- Now, run the script from your terminal and check your browser

```
(flask2env) C:\Users\unan\Desktop\flask2>python main.py
 * Serving Flask app "main" (lazy loading)
 * Environment: production
   WARNING: Do not use the development server in a production environment.
   Use a production WSGI server instead.
 * Debug mode: on
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 259-066-325
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```



welcome to Flask exercise 1

Modify the HTML file

- Add some more text and image

```
<h1>welcome to Flask exercise 1</h1>
<p>Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod
<img src="https://images.unsplash.com/photo-1563169372-eb64c121f9dc?ixlib=r
```

127.0.0.1:5000

welcome to Flask exercise 1

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et d
ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu
mollit anim id est laborum.

Template Variables

- Using the render_template function we can directly render an HTML file with our Flask web app
- But we haven't leveraged the power of python at all yet
- We want a way to be able to use Python code in our app, changing and updating variables and logic, and then send that information to the template
- We can use the **Jinja** template engine to do this

https://www.udemy.com/python-and-flask-bootcamp-create-websites-using-flask/learn/lecture/9340994?start=0#questions

Jinja

- Jinja templating will let us directly insert variables from our Python code to the HTML file.
- The syntax for inserting a variable is;
  - {{variable_name}}
- We can pass in python strings, lists, dictionaries, and more into the templates
- We set parameters (of our choosing) in the render_template function and then use the {{}} syntax to insert them in the template

https://www.udemy.com/python-and-flask-bootcamp-create-websites-using-flask/learn/lecture/9340994?start=0#questions

Exercise 2

- Create **exercise2.py** file under flask_example folder
- Create **exercise2.html** file under templates folder

```
from flask import Flask, render_template
app = Flask (__name__)

@app.route ('/')
def index ():
    name="Mahmut"
    return render_template('exercise2.html',my_name=name)

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

*** You can use the same variable name also; **name=name**

```html
<!DOCTYPE html>
<html lang="en" dir="ltr">
  <head>
    <meta charset="utf-8">
    <title></title>
  </head>
  <body>
    <h1>Exercise 2 </h1>
    <h2>My name is {{my_name}}</h2>
  </body>
</html>
```

127.0.0.1:5000

Exercise 2

My name is Mahmut

Exercise 3

- Pass your name variable and a list that contains the letters of that name
- Display the letters also

Exercise 2

My name is Mahmut

['M', 'a', 'h', 'm', 'u', 't']

Template Control Flow

- With Jinja templating we can pass variables using {{variable}} syntax
- We also have access to control flow syntax in our templates such as for loops and if statements
- We will use {%. %} syntax for these
- Let’s say we are passing a list as a parameter to the html file and we want to display each item of this list as a bulleted HTML list

https://www.udemy.com/python-and-flask-bootcamp-create-websites-using-flask/learn/lecture/9340994?start=0#questions

```
<ul>
    {% for item in mylist%}
    <li> {{item}} </li>
    {% endfor %}
</ul>
```

Exercise 4

- Create a list in your .py file
  - days= ["Sunday", "Monday", "Tuesday","Wednesday","Thursday", "Friday", "Saturday"]
- In your html file, display these list as bulletin. If it is a weekend day, make it bold

Exercise 4

- **Sunday**
- Monday
- Tuesday
- Wednesday
- Thursday
- Friday
- **Saturday**

```python
from flask import Flask, render_template
app = Flask (__name__)

@app.route ('/')
def index ():
    days= ["Sunday", "Monday", "Tuesday","Wednesday","Thursday", "Friday", "Satu
    return render_template('exercise4.html', days=days)


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

```html
<body>
  <h1>Exercise 4</h1>
  <ul>
    {% for day in days%}
     {%if day == "Sunday" or day =="Saturday" %}
        <li> <b>{{day}} </b> </li>
    {% else %}
         <li> {{day}}</li>
    {% endif %}
    {% endfor %}
  </ul>
```

Exercise 4

- **Sunday**
- Monday
- Tuesday
- Wednesday
- Thursday
- Friday
- **Saturday**

Template Inheritance

- We know we can create view functions that directly link to an HTML template
- But that still means we need to make an HTML template for every page
- Usually pages across a web application already share a lot of features (e.g. navigation bar, footer, etc…)
- A great solution is template inheritance
- We set up a base.html template file with the re-usable aspects of our site
- Then we use {% extend “base.html” %} and {% block %} statements to extend these re-usable aspects to another pages

https://www.udemy.com/python-and-flask-bootcamp-create-websites-using-flask/learn/lecture/9340994?start=0#questions

Filters

- Filters are a great way to quickly change/edit a variable passed to a template
- {{ variable | filter }}

- For example
- {{ name }}
  - mahmut
- {{ name | capitalize }}
  - Mahmut

https://www.udemy.com/python-and-flask-bootcamp-create-websites-using-flask/learn/lecture/9341006#questions

Exercise 5

- Create 3 html files
  - base.html
  - home.html
  - contact-us.html
- Create exercise5.py file

base.html

- Add a title → this tittle will be display on all pages
- Add the bootstrap link (https://getbootstrap.com/)
- Create a navigation bar in the body

```
    <title>My Company</title>
    <script src="https://code.jquery.com/jquery-3.3.1.slim.min.js" integrity="s
    <script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.14.7/umd/po
    <script src="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/js/bootstra

  </head>
  <body>
    <nav class="nav">
    <a class="nav-link active" href="/">Home Page </a>
    <a class="nav-link active" href="/contact-us"> Contact US</a>
    </nav>
  {% block content %}

  {% endblock %}
```

exercise5.py

```python
from flask import Flask, render_template
app = Flask (__name__)

@app.route ('/')
def index ():
    return render_template('home.html')



@app.route ('/contact-us')
def info ():
    return render_template('contact-us.html')

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

home.html

```
{% extends "base.html"%}

{% block content %}
<h1>This is the Home Page</h1>

{% endblock %}
```

127.0.0.1:5000

Home Page Contact US

This is the Home Page

contact-us.html

```
{% extends "base.html"%}

{% block content %}
<h1>Contact US</h1>

{% endblock %}
```

127.0.0.1:5000/contact-us

Home Page Contact US

Contact US

Template Forms

- In the HTML lectures we learned how to create HTML forms for users to supply information
- Now, we will learn how we can connect our Flask application to these forms

Forms with Flask

- We will create a basic form using our flask python scripts.
- We will be using/learning **flask_wtf** and **wtforms** packages
- Initially we will configure a secret key for security
- Then, we will crea a WTForm Class
  - Create Fields for each part of the form
- Later on, we will set up a View Function
  - Add methods= [‘GET’, ‘POST’]
  - Create an instance of Form Class
  - Handle Form submission

https://www.udemy.com/python-and-flask-bootcamp-create-websites-using-flask/learn/lecture/9345780#questions

Exercise 7

- Create a folder = **flask3**
- Create our flask environment inside that folder
- Create a **main.py** file under flask3 folder
- Create a subfolder = **templates**
- Create a **home.html** file under the templates folder

https://www.udemy.com/python-and-flask-bootcamp-create-websites-using-flask/learn/lecture/9345780#questions

main.py

- We will start importing the libraries that we use

```
from flask import Flask, render_template
from flask_wtf import FlaskForm
from wtforms import StringField,SubmitField
```

- Set up your application and your applications secret key

```
app =Flask (__name__)

app.config['SECRET_KEY'] = 'oursecretkey'
```

https://www.udemy.com/python-and-flask-bootcamp-create-websites-using-flask/learn/lecture/9345780#questions

main.py / 2

- Create a Form Class and create the fields

```
class MyForm(FlaskForm):
    customer = StringField('What is your customer id?')
    submit = SubmitField('Submit')
```

- Set up the view function and select how message will be delivered

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

main.py / 3

- Create the view function and run the script

```
@app.route('/', methods=['GET','POST'])
def index():
    customer =False
    form = MyForm()
    if form.validate_on_submit():
        customer = form.customer.data
        form.customer.data= ''
    return render_template('main.html', form=form, customer=customer)

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

https://www.udemy.com/python-and-flask-bootcamp-create-websites-using-flask/learn/lecture/9345780#questions

main.html

- Let’s check whether the customer id is empty or not (the default was false)

```
<p>
{% if customer %}
The customer id you entered is {{customer}}
Update it in the form below:
{% else %}
Please enter your customer id:
{% endif %}
</p>
```

main.html / 2

- create a POST method and create the form

```
<form method = "POST">
  {{ form.hidden_tag() }}
  {{ form.customer.label }} {{ form.customer() }}
  {{ form.submit() }}
</form>
```

- Since we send a form object (form), we will call that attributes (customer and submit)
- We can to inherit some other attributes even though we haven’t used them yet. That’s why we are using hidden_tag() feature

Views

127.0.0.1:5000

Please enter your customer id:

What is your customer id? Submit

127.0.0.1:5000

The customer id you entered is unan Update it in the form below:

What is your customer id? Submit

127.0.0.1:5000

Please enter your customer id:

What is your customer id? Submit

Form Fields

- Every possible HTML form field has a corresponding wtforms class you can import
- wtforms also has validators you can easily insert
- Validators can perform checks on the form data, such as requiring a field to be filled
- We will also use Flask's session object to grab the information provided in the form and pass it to another template

https://www.udemy.com/python-and-flask-bootcamp-create-websites-using-flask/learn/lecture/9345780#questions

Exercise 8

- Create the following files;

  - exercise8.py (under flask3 folder)
  - index.html (under templates folder)
  - thankyou.html (under templates folder)

exercise8.py

- We will import quite a few libraries;
  - from flask import Flask, render_template, session, redirect, url_for
  - from flask_wtf import FlaskForm
  - from wtforms import StringField,SubmitField,BooleanField,DateTimeField,RadioField,SelectField,TextField, TextAreaField
  - from wtforms.validators import DataRequired

```
from flask import Flask, render_template, session,redirect,url_for
from flask_wtf import FlaskForm
from wtforms import (StringField,SubmitField,BooleanField,DateTimeField,
                     RadioField,SelectField,TextField, TextAreaField)
from wtforms.validators import DataRequired
```

exercise8.py / 2

- Configure secret key and the form class
- We will try to add a few different attributes
- We are using validators to select the required fields

```
app.config['SECRET_KEY'] = 'anothersecretkey'

class InfoForm(FlaskForm):
    department = StringField  (' What is your department?', validators=[DataRequired()])
    graduated = BooleanField ('Did you Graduate?')
    degree = RadioField (' Please choose your degree: ', choices=[('level_one','Undergrad'),('level_two','Graduate')])
    job_choice = SelectField ('Where do you want to work?', choices=[('job1','Industry'),('job2','Academia')])
    feedback = TextAreaField()
    submit=SubmitField('Submit')
```

exercise8.py / 3

- Let’s create our view function
- We will use session to send all the data, session works like a dictionary
- We will use url_for in python to add the link

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

    form = InfoForm()
    if form.validate_on_submit():
        session['department']=form.department.data
        session['graduated']=form.graduated.data
        session['degree']=form.degree.data
        session['job_choice']=form.job_choice.data
        session['feedback']=form.feedback.data
        return redirect(url_for('thankyou'))

    return render_template ('index.html', form = form)
```

exercise8.py / 4

- Create a thank you page and run the app

```
@app.route('/thankyou')
def thankyou():
    return render_template ('thankyou.html')

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

index.html

```
<body>
  <h1>Welcome to the CS survey</h1>
  <form  method="post">
    {{form.hidden_tag()}}
    {{form.department.label}} {{form.department}}
    <br>
    {{form.graduated.label}} {{form.graduated}}
    <br>
    {{form.degree.label}} {{form.degree}}
    <br>
    {{form.job_choice.label}} {{form.job_choice}}
    <br>
    {{form.feedback.label}} {{form.feedback}}
    <br>
    {{form.submit()}}

  </form>

</body>
```

thankyou.html

```
<body>
  <h1> Thank, please check your information below</h1>
  <ul>
    <li> Department : {{session['department']}}</li>
    <li> Graduated : {{session['graduated']}}</li>
    <li> Degree : {{session['degree']}}</li>
    <li> Job Choice : {{session['job_choice']}}</li>
    <li> Feedback : {{session['feedback']}}</li>
  </ul>

</body>
```

Views

127.0.0.1:5000

Welcome to the CS survey

What is your department? Math

Did you Graduate? ☑

Please choose your degree:

- ○ Undergrad
- ◉ Graduate

Where do you want to work? Academia

Feedback no feedback

Submit

127.0.0.1:5000/thankyou

Thank you, please check your information below

- Department : Math
- Graduated : True
- Degree : level_two
- Job Choice : job2
- Feedback : no feedback

SQL Databases with Flask

- We now understand how to grab user information through Forms with Flask
- Let’s build on top of this understanding by linking our Flask applications to a database so we can save user information
- SQL allows us to store data in a tabular format
- I assume everybody know SQL

https://www.udemy.com/python-and-flask-bootcamp-create-websites-using-flask/learn/lecture/9345780#questions

Python – Flask & SQL

- Python and Flask can connect to a variety of SQL Database engines, including PostgreSQL, MySQL, SQLite, and many more
- SQLite is a simple SQL database engine that comes with Flask and can handle all our needs
- SQLite can actually scale quite well for basic applications (100,000 hits per day)
- To connect python, Flask, and SQL together we will need an ORM (Object Relational Mapper)

https://www.udemy.com/python-and-flask-bootcamp-create-websites-using-flask/learn/lecture/9345780#questions

Python – Flask & SQL

- An ORM will allow us to directly use Python instead of SQL syntax to create, edit, update, and delete from our database
- The most common ORM for Python is **SQL Alchemy**
- Flask-SQLAlchemy is an extension that allows for an easy connection of Flask with SQLAlchemy
  - **`pip install Flask-SQLAlchemy`**

https://www.udemy.com/python-and-flask-bootcamp-create-websites-using-flask/learn/lecture/9345780#questions

Databases with Flask

- To begin working with Databases, we'll do the following;
  - Set up SQLite Database in a Flask App
  - Create a Model in Flask App
  - Perform basic **CRUD** on our model (CRUD stands for create, read, update, delete)
- To create a SQLITE database
  - Create a Flask App
  - Configure Flask App for SQLAlchemy
  - Pass our application into the SQLAchemy class call

https://www.udemy.com/python-and-flask-bootcamp-create-websites-using-flask/learn/lecture/9345780#questions

Create a Model in Flask

- Models ~= table in SQL database
- You do not need to create the table manually with SQL
- Instead we simply create a Model class in Python that generates the table for us
- Similar to creating a FlaskForm, for models:
  - Create a model class
  - Inherit from db.Model
  - Optionally provide a table name
  - Add in table columns as attributes
  - Add in methods for **__init__** and **__repr__**

https://www.udemy.com/python-and-flask-bootcamp-create-websites-using-flask/learn/lecture/9345780#questions

Exercise 1

- We will show our CRUD operations being performed manually in a .py script
- Keep in mind, this is just to understand the syntax, typically a lot of this will be automated with Flask
- Create a new folder => lecture12
- Create the Flask environment
- Inside the lecture12 folder create a **main.py** and **setupdatabase.py** files

https://www.udemy.com/python-and-flask-bootcamp-create-websites-using-flask/learn/lecture/9345780#questions

main.py

```
import os
from flask import Flask
from flask_sqlalchemy import SQLAlchemy

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

__file__ refers to main.py

abspath → absolute path → provides the full directory path

main.py

- set up the database location and configure the track modification settings (we don't want to track every single modification)
- create the database and pass our app in

```
app = Flask(__name__)

app.config['SQLALCHEMY_DATABASE_URI']='sqlite:///'+os.path.join(basedir,'data.sqlite')
app.config['SQLALCHEMY_TRAC_MODIFICATIONS']=False

db=SQLAlchemy(app)
```

main.py

- Now, we will create our model class
- A default name will be provided, however, we can overwrite the name
- We will create the initialization function and a function to represent

```
class Student(db.Model):
    __tablename__="students"

    id= db.Column(db.Integer, primary_key=True)
    name= db.Column(db.Text)
    grade=db.Column(db.Integer)

    def __init__(self,name,grade):
        self.name=name
        self.grade=grade

    def __repr__(self):
        return f"Student {self.name} got {self.grade} on midterm exam"
```

setupdatabase.py

- Initially we will import our db, and Student class from main.py file
  - from main import db,Student
- Create all the tables
  - db.create_all()

- We will create two objects mahmut and sam and assign some values. If we try to print them, initially they will print None

setupdatabase.py

```
from main import db,Student

db.create_all()

mahmut = Student('Mahmut',100)
sam = Student ('Sam', 105)

#initially they will print none
print (sam.id)
print(mahmut.id)

db.session.add_all([mahmut,sam])

db.session.commit()

print(mahmut.id)
print(sam.id)
```

Command Window

- Go to command window and run the setupdatabase.py
  - python setupdatabase.py

```
(myflak4env) C:\Users\umahm\Desktop\flask4>pyth
C:\Users\umahm\Anaconda3\envs\myflak4env\lib\si
cationWarning: SQLALCHEMY_TRACK_MODIFICATIONS a
lt in the future.  Set it to True or False to s
  'SQLALCHEMY_TRACK_MODIFICATIONS adds signific
None
None
1
2
```

- Notice that, some files are created for you → data.sqlite…

How about CRUD?

- Let’s create a new file and call it to **crud.py**
- import the database
  - `from main import db,Student`
- To add a new record

```
from main import db,Student


new_student = Student ('John', 90)
db.session.add(new_student)
db.session.commit()
```

crud.py / 2

- Reading all students or a specific student using the id

```
all_students = Student.query.all() #List all all_students
print (all_students)

# Select the student by id
first_student = Student.query.get(1)
print(first_student.name)
```

- Display students whose grade is bigger than 85

```
# Filters
student_pass = Student.query.filter(Student.grade>=85)
print(student_pass.all())
```

crud.py / 3

- Update an entry or delete an entry

```
#Update
first_student = Student.query.get(1)
first_student.grade = 105
db.session.add(first_student)
db.session.commit()


#delete
second_student = Student.query.get(2)
db.session.delete(second_student)
db.session.commit()
```

crud.py / 4

- Let’s check the changes

```
all_students = Student.query.all() #List all all_students
print (all_students)
```

- Run the crud.py file and observe the changes

HW 3

- You will connect your exercise 1 with a user interface (HTML file, forms…etc)

Database Migrations

- So far, we have learned the creation and use of a database, but how about making updates to an existing database as the application needs change or grow.
  - Such as, adding a new column
- It is a tricky job because relational databases are centered around structured data.
  - when the structure changes the data that is already in the database needs to be ***migrated*** to the modified structure.

https://blog.miguelgrinberg.com/post/the-flask-mega-tutorial-part-iv-database

Migrate

```
pip install flask-migrate
```

- We have already installed it (inside the requirements.txt)

- This will give us the opportunity to update our Model class and SQL database
- There are some commands that we use at the command line;

- Set the FLASK_APP environment variable
  - MacOS/Linux

    export FLASK_APP=myapp.py
  - Windows

    set FLASK_APP=myapp.py

flask db init

- Set up the migrations directory

flask db migrate –m “put a message here”

- set up the migration file

flask db upgrade

- update the database with migration

https://www.udemy.com/python-and-flask-bootcamp-create-websites-using-flask/learn/lecture/9345780#questions

Exercise 2

- In this exercise we will be updating our Exercise 1
- Let's start with the main.py file
- First import the Migrate from flask_migrate library

```
from flask_migrate import Migrate
```

- Connect the app and the db with the migrate

```
app = Flask(__name__)

app.config['SQLALCHEMY_DATABASE_URI']='sqlite:///'+os.path.join(basedir,'data.sqlite')
app.config['SQLALCHEMY_TRAC_MODIFICATIONS']=False

db=SQLAlchemy(app)
Migrate(app,db)
```

Command line

- After saving the modifications to main.py, open the command line

```
export FLASK_APP=main.py
```

for windows users

```
// set FLASK_APP=myapp.py

flask db init
flask db migrate -m “I did something”
flask db upgrade
```

Let’s modify our database

- open main.py file again and insert one more column to our database

```
class Student(db.Model):
    __tablename__="students"

    id= db.Column(db.Integer, primary_key=True)
    name= db.Column(db.Text)
    grade=db.Column(db.Integer)
    attendance = db.Column(db.Integer)

    def __init__(self,name,grade):
        self.name=name
        self.grade=grade
        self.attendance=attendance

    def __repr__(self):
        return f"Student {self.name} got {self.grade} on midterm exam, attendance = {self.attendance}"
```

Command line

- after saving the main.py file, let’s migrate it again

```
flask db migrate -m “added
attendance column”
flask db upgrade
```

Exercise 3

- Create a simple app for a bookstore;
  - Book model
    - **Name of the book**
    - **Author**
    - **ISBN**
    - **Number of Pages**
- Implement the CRUD operations
- Implement the features for migration
- Update the database and add **publisher** column