<codoa codo/>

Agencia de Aprendizaje a lo largo de la vida

Flask
Clase 32

Flask: Vistas y Modelos

BA

Les damos la bienvenida

Vamos a comenzar a grabar la clase

¿Qué es la Vista?

La vista entre otras cosas contiene el código fuente necesario para responder a las acciones que el usuario solicita y servir como puente de interacción entre las vistas y el modelo de datos. Normalmente podemos tener más de una vista dependiendo la organización de nuestro proyecto y la magnitud de este.

La vista en este caso con FLASK y desarrollando una API-Rest será la encargada de gestionar las rutas de enlace y determinar cuál será la respuesta en cada url así como también la gestión de los datos que obtiene o solicita al modelo.

Creación de métodos en la vista

En el archivo views.py de la aplicación vamos a crear distintas funciones que permitirán gestionar el CRUD de tareas. Posteriormente en el archivo run.py se tendrá que asociar dichas funciones, con las URLs y métodos HTTP que queremos escuchar

app/views.py – Traer todas las tareas pendientes

```python
def get_pending_tasks():
    tasks = [
        {
            'id':1,
            'nombre':'Tarea 1',
            'descripcion':'Tarea 1 Desc',
            'completada': False,
            'activo': True,
            'fecha_creacion': '2024-01-01'
        },
        {
            'id':2,
            'nombre':'Tarea 2',
            'descripcion':'Tarea 2 Desc',
            'completada': False,
            'activo': True,
            'fecha_creacion': '2024-01-01'
        }
    ]
    return jsonify(tasks)
```

run.py

```python
app.route('/api/tasks/pending/', methods=['GET'])(get_pending_tasks)
app.route('/api/tasks/completed/', methods=['GET'])(get_completed_tasks)
app.route('/api/tasks/archived/', methods=['GET'])(get_archived_tasks)
```

Creación de métodos en la vista

app/views.py – Traer una tarea

```
def get_task(task_id):
    task = {
        'id':task_id,
    }
    return jsonify(task)
```

run.py

```
app.route('/api/tasks/fetch/<int:task_id>', methods=['GET'])(get_task)

app.route('/api/tasks/create/', methods=['POST'])(create_task)
```

app/views.py – Crear una tarea

```
from flask import jsonify, request
...

def create_task():
    #datos recibidos en formato json
    data = request.json
    return jsonify({'message': 'Task created successfully','data':data}), 201
```

Creación de métodos en la vista

app/views.py – Actualizar una tarea

```
def update_task(task_id):
    #datos recibidos en formato json
    data = request.json
    return jsonify({'message': 'Task updated
successfully','data':data,'id':task_id})
```

run.py

```
app.route('/api/tasks/update/<int:task_id>', methods=['PUT'])(update_task)
app.route('/api/tasks/complete/set/<int:task_id>', methods=['PUT'])(set_complete_task)
app.route('/api/tasks/complete/reset/<int:task_id>', methods=['PUT'])(reset_complete_task)
app.route('/api/tasks/archive/<int:task_id>', methods=['DELETE'])(archive_task)
```

app/views.py – Archivar una tarea y completar / marcar como pendiente

```
def archive_task(task_id):
    return jsonify({'message': 'Task archived successfully','id':task_id})
def set_complete_task(task_id):
    return jsonify({'message': 'Task updated successfully','id':task_id})
def reset_complete_task(task_id):
    return jsonify({'message': 'Task updated successfully','id':task_id})
```

¿Qué es un Modelo?

La capa del Modelo en la arquitectura MVC es aquella que se encarga de trabajar con los datos; por lo que normalmente dentro del modelo encontraremos mecanismos para acceder a la información y actualizar los datos en la fuente de almacenamiento, que bien podría ser una base de datos relacional o no relacional.

Creación del Modelo

En el archivo app/models.py se van a definir cada modelo que contendrá los campos y métodos que pueden ser utilizados para interactuar con la base de datos.
Por lo que vamos a utilizar la implementación de una clases, definiremos los atributos de la clase correspondientemente con los campos de la tabla “tasks” de nuestra base de datos.

```python
from app.database import get_db

class Task:
    def __init__(self, id_task=None, nombre=None, descripcion=None, fecha_creacion=None, completada=None, activa=None):
        self.id_task = id_task
        self.nombre = nombre
        self.descripcion = descripcion
        self.fecha_creacion = fecha_creacion
        self.completada = completada
        self.activa = activa
```

Guardado de imágenes

La forma más recomendada para guardar imágenes en nuestro backend con Flask es almacenando los archivos en un directorio local de nuestro backend.

Aquí hay un par de links que guían este proceso:

Documentación oficial de Flask:
https://flask.palletsprojects.com/en/1.1.x/patterns/fileuploads/

Medium (Basado en el post de abajo):
https://medium.com/@brodiea19/flask-sqlalchemy-how-to-upload-photos-and-render-them-to-your-webpage-84aa549ab39e

Post original:
https://blog.miguelgrinberg.com/post/handling-file-uploads-with-flask

Discusiones en Stackoverflow respecto a este tema:
https://stackoverflow.com/a/38248933
https://stackoverflow.com/questions/3748/storing-images-in-db-yea-or-nay/3751#3751

Creación del Modelo

Método para traer listado de tareas

```python
@staticmethod
def get_all_pending():
    return Task.__get_tasks_by_query(
        """ SELECT * FROM tareas WHERE activa = true AND completada = false
            ORDER BY fecha_creacion DESC""")

@staticmethod
def get_all_completed():
    return Task.__get_tasks_by_query(
        """ SELECT * FROM tareas WHERE activa = true AND completada = true
            ORDER BY fecha_creacion DESC""")

@staticmethod
def get_all_archived():
    return Task.__get_tasks_by_query(
        """ SELECT * FROM tareas WHERE activa = false
            ORDER BY fecha_creacion DESC""")
```

```python
@staticmethod
def __get_tasks_by_query(query):
    db = get_db()
    cursor = db.cursor()
    cursor.execute(query)
    rows = cursor.fetchall()

    tasks = []
    for row in rows:
        tasks.append(
            Task(
                id_task=row[0],
                nombre=row[1],
                descripcion=row[2],
                fecha_creacion=row[3],
                completada=row[4],
                activa=row[5]
            )
        )
    cursor.close()
    return tasks
```

Creación del Modelo

Método para traer una tarea

```
    @staticmethod
    def get_by_id(id_task):
        db = get_db()
        cursor = db.cursor()
        cursor.execute("SELECT * FROM tareas WHERE id = %s", (id_task,))
        row = cursor.fetchone()
        cursor.close()

        if row:
            return Task(
                id_task=row[0],
                nombre=row[1],
                descripcion=row[2],
                fecha_creacion=row[3],
                completada=row[4],
                activa=row[5]
            )
        return None
```

Creación del Modelo

Método para guardar/actualizar una tarea

```
    def save(self):
        db = get_db()
        cursor = db.cursor()
        if self.id_task: # Actualizar Tarea existente
            cursor.execute(
                """UPDATE tareas
                SET nombre = %s, descripcion = %s, completada = %s, activa = %s
                WHERE id = %s""",
                (self.nombre, self.descripcion, self.completada, self.activa, self.id_task))
        else: # Crear Tarea nueva
            cursor.execute(
                """INSERT INTO tasks
                (nombre, descripcion, fecha_creacion, completada, activa)
                VALUES (%s, %s, %s, %s, %s)""",
                (self.nombre, self.descripcion, self.fecha_creacion, self.completada, self.activa))
            self.id_task = cursor.lastrowid
        db.commit()
        cursor.close()
```

Creación del Modelo

Método para eliminar (Borrado lógico) una tarea

```
    def delete(self):
        db = get_db()
        cursor = db.cursor()
        cursor.execute("UPDATE tareas SET activa = false WHERE id = %s", (self.id_task,))
        db.commit()
        cursor.close()
```

Método serializador de una tarea

Este método nos permitirá representar la instancia de un objeto de la clase Task, como un diccionario. Esto nos resultará conveniente para que desde la vista podamos hacer una conversión directa desde una representación de diccionario a formato JSON.

```
    def serialize(self):
        return {
            'id': self.id_task,
            'nombre': self.nombre,
            'descripcion': self.descripcion,
            'fecha_creacion': self.fecha_creacion.strftime('%Y-%m-%d'),
            'completada': self.completada,
            'activa': self.activa
        }
```

Completamos código de la vista

app/views.py – Traer todas las tareas

```python
from app.models import Task

def get_pending_tasks():
    tasks = Task.get_all_pending()
    return jsonify([task.serialize() for task in tasks])

def get_completed_tasks():
    tasks = Task.get_all_completed()
    return jsonify([task.serialize() for task in tasks])

def get_archived_tasks():
    tasks = Task.get_all_archived()
    return jsonify([task.serialize() for task in tasks])
```

Completamos código de la vista

app/views.py – Traer una tarea

```
def get_task(task_id):
    task = Task.get_by_id(task_id)
    if not task:
        return jsonify({'message': 'Task not found'}), 404
    return jsonify(task.serialize())
```

Completamos código de la vista

app/views.py – Crear una tarea

```
def create_task():
    data = request.json
    new_task = Task(
        nombre=data['nombre'],
        descripcion=data['descripcion'],
        fecha_creacion=date.today().strftime('%Y-%m-%d'),
        completada=False,
        activa=True
    )
    new_task.save()
    return jsonify({'message': 'Task created successfully'}), 201
```

Completamos código de la vista

app/views.py – Actualizar una tarea

```
def update_task(task_id):
    task = Task.get_by_id(task_id)
    if not task:
        return jsonify({'message': 'Task not found'}), 404

    data = request.json
    task.nombre = data['nombre']
    task.descripcion = data['descripcion']
    task.save()
    return jsonify({'message': 'Task updated successfully'})
```

Completamos código de la vista

app/views.py – Completar una tarea

```python
def __complete_task(task_id, status):
    task = Task.get_by_id(task_id)
    if not task:
        return jsonify({'message': 'Task not found'}), 404

    task.completada = status
    task.activa = True
    task.save()
    return jsonify({'message': 'Task updated successfully'})

def set_complete_task(task_id):
    return __complete_task(task_id, True)

def reset_complete_task(task_id):
    return __complete_task(task_id, False)
```

Completamos código de la vista

app/views.py – Eliminar una película

```
def archive_task(task_id):
    task = Task.get_by_id(task_id)
    if not task:
        return jsonify({'message': 'Task not found'}), 404

    task.delete()
    return jsonify({'message': 'Movie deleted successfully'})
```

Integración de Backend - Frontend

Un problema que puede ser común a la hora de integrar aplicaciones frontend y backend suele esta relacionado con CORS (Cross-Origin Resource Sharing), indica que tu aplicación cliente (que se ejecuta en el origen ej http://127.0.0.1:5500) está tratando de hacer una solicitud a un servidor en un origen diferente (http://127.0.0.1:5000) y el servidor no está configurado para permitir solicitudes desde el origen del cliente. Por lo que desde nuestro proyecto backend vamos a configurarlo para que solucionar este error.

En el ambiente virtual del proyecto vamos a instalar la librería flask-cors. (No te olvides de actualizar el archivo requirements.txt)

```
pip install flask-cors
```

run.py

En el archivo run.py vamos a agregar el siguiente código.

```
from flask_cors import CORS
...

# Inicializar la base de datos con la aplicación Flask
init_app(app)
#permitir solicitudes desde cualquier origen
CORS(app)
```

No te olvides de completar la asistencia y consultar dudas

Recordá:

- Revisar la Cartelera de Novedades.
- Hacer tus consultas en el Foro.

TODO EN EL AULA VIRTUAL