Lista de tareas con Flask

Created	@February 22, 2023 12:08 PM
Class	Python sin fronteras
Profe	Nico Schurman
Materials	https://www.udemy.com/course/python-sin-fronteras-html-css-mysql/learn/lecture/22223100#questions
Reviewed	☐
Tipo	Proyecto

▼ Preparación

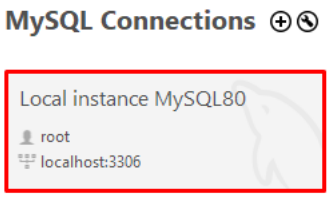
1. Primero creamos la carpeta e instalamos y activamos el entorno virtual.

```
mkdir ltflask
cd ltflask
touch lt.py
python -m venv venv
source venv/scripts/activate
```

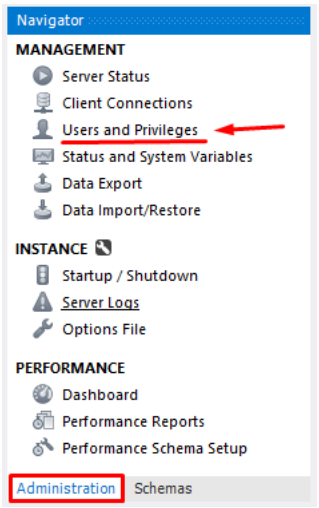
2. Ahora instalamos las herramientas que vamos a ocupar para este proyecto.

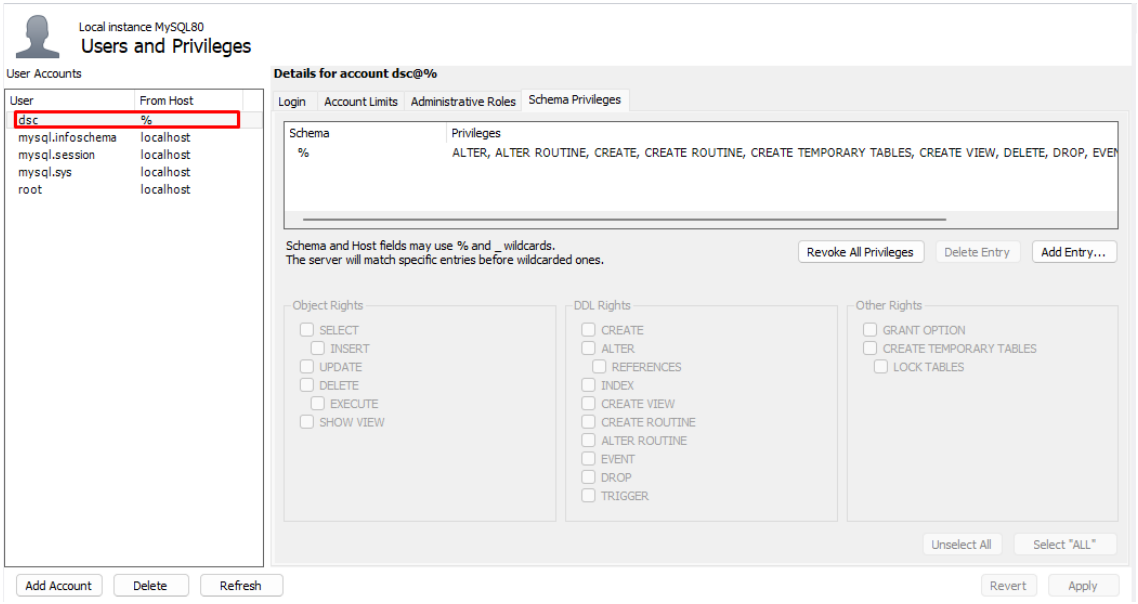
```
pip install werkzeug #
pip install mysql-connector-python #conector SQL
pip install Flask #framework flask
```

3. Vamos a MySQL Workbench y asignamos todos los permisos para nuestro usuario en base de datos para poder crear libremente tablas usando un usuario distinto al de root.
    i. Vamos a MySQL Workbench, y al usuario de root.

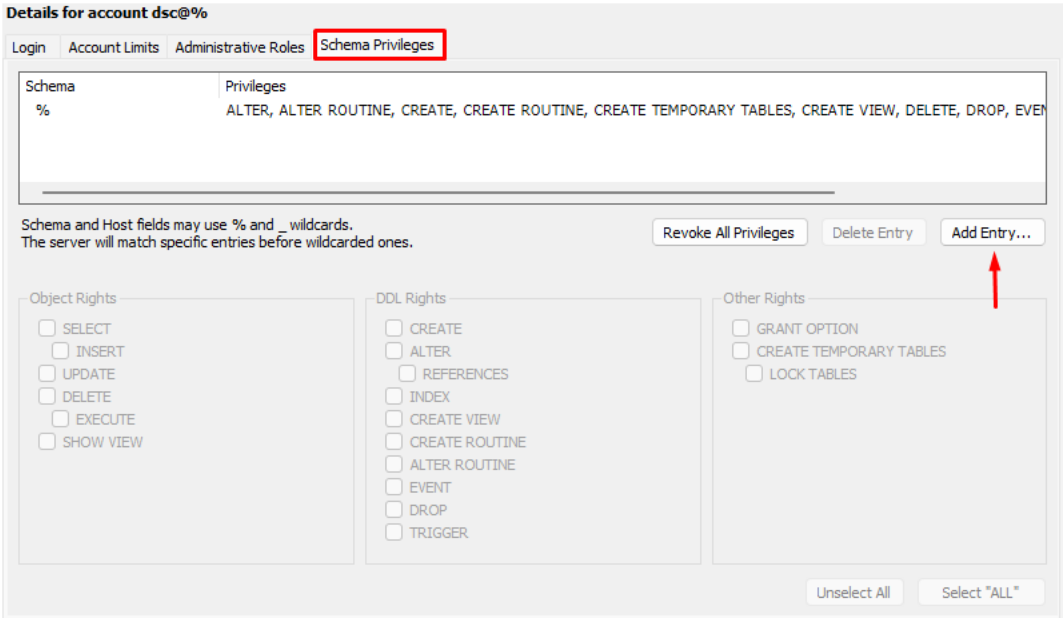


    ii. Vamos a Administration/'Users and Privileges' y seleccionamos nuestro usuario.

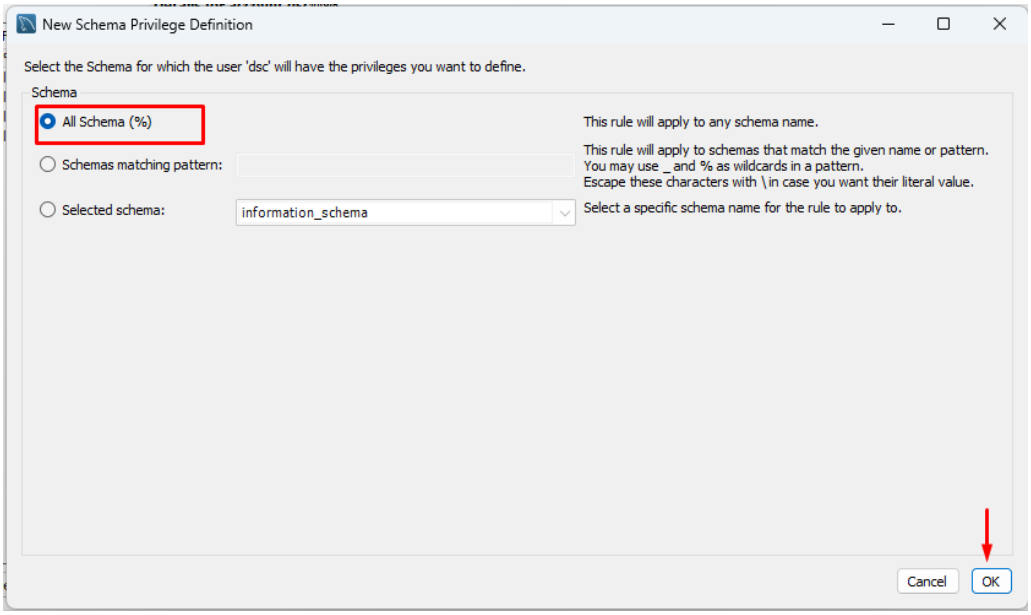




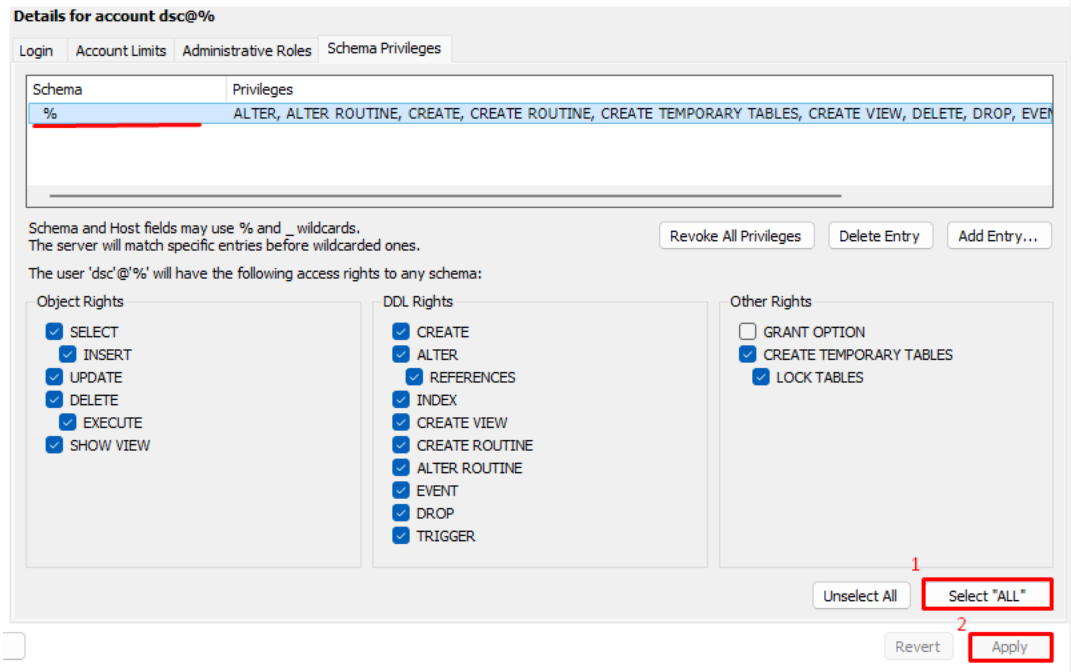
iii. Vamos a la pestaña Schema Privileges, y pinchamos “Add Entry”.



iv. Seleccionamos todos los esquemas y damos click en “ok”.



v. Luego selecciono la entrada recien creada, selecciono todos los privilegios y aplico los cambios.



▼ Orden de carpetas

1. Crearemos dentro de la carpeta del proyecto, una carpeta llamada “todo”.

```
mkdir todo
```

2. Dentro crearemos un archivo llamado ‘__init__.py’. Este archivo hará que python interprete la carpeta de ‘todo’ como si este fuera un modulo.

```
cd todo
touch __init__.py
```

3. Volvemos a la carpeta del proyecto, configuramos ambiente de desarrollo en Flask y asignamos la app a la carpeta todo.

```
cd ..
export FLASK_APP=todo
export FLASK_ENV=development
```

▼ Edición de ‘__init__.py’

```
import os # (operative system) usamos esta libreria para acceder a las variables de entorno.
from flask import Flask #importamos flask.

#para trabajar con esta separacion de modulos debemos crear la funcion create_app.
def create_app(): #servirá para hacer testing o para crear instancias de la app.
    app = Flask(__name__) #Toda aplicación creada en Flask es una instancia de la clase Flask.

#Configuramos nuestra aplicación.
    app.config.from_mapping( #Vamos a definir variables de configuracion con from_mapping.
        SECRET_KEY = 'mikey', #Llave(cookie) a utilizar para definir las sesiones.
    #definimos el host, password, user y ddbb a partir de variables de entorno.
        DATABASE_HOST = os.environ.get('FLASK_DATABASE_HOST'),
        DATABASE_PASSWORD = os.environ.get('FLASK_DATABASE_PASSWORD'), #
        DATABASE_USER = os.environ.get('FLASK_DATABASE_USER'),
        DATABASE = os.environ.get('FLASK_DATABASE'),
    )
#Creamos una ruta de pruebas.
```

```
    @app.route('/hola')
    def hola():
        return 'chanchito feliz'

    return app
```

▼ Creando conexión a BBDD

1. Dentro de la carpeta ‘todo’ vamos a crear un archivo llamado ‘db.py’.

```
cd todo
touch db.py
```

2. Definimos la conexión que se hará a la base de datos:

```
import mysql.connector #importamos el conector de mysql
import click #importamos click para poder para poder ejecutar comandos en la terminal
from flask import current_app, g #current_app mantiene la app que estamos ejecutando
                                 #g es una variable que está en toda nuestra app. La usaremos para almacenar el usuario.
from flask.cli import with_appcontext #con appcontext podemos acceder a las variables de la config.
from .schema import instructions #Lo creamos nosotros. Contiene los scripts para poder crear nuestra bbdd.

#Definimos una funcion que nos permita obtener nuestra ddbb y el cursor dentro de nuestra app.
def get_db():
    if 'db' not in g: #Si no se encuentra el atributo 'db' dentro de g
        g.db = mysql.connector.connect( #Vamos a generar una propiedad dentro de g que contendrá la conexión a la bbdd.
            host = current_app.config['DATABASE_HOST'], #Aquí usaremos la propiedad de current_app para definir las credenciales ir
            user = current_app.config['DATABASE_USER'],
            password = current_app.config['DATABASE_PASSWORD'],
            database = current_app.config['DATABASE'],
        )
        g.c = g.db.cursor(dictionary = True) #agregamos la obtención del cursor con acceso a las propiedades en forma de diccionari

    return g.db, g.c #Así al llamar a get_db() vamos a obtener la bbdd y el cursor.

#Definimos ahora una función que nos cierre la conexión de la bbdd cada vez que se realice una petición.
#La idea es que no debemos dejar esa conexión abierta con cada llamado.
def close_db(e = None):
    db = g.pop('db', None) #se encarga de obtener la variable db del objeto g que es un objeto global que se utiliza para almacenar
    #La función pop() busca la clave db en el diccionario g. Si se encuentra la clave db, se devuelve su valor y se elimina la clav
    if db is not None: #Luego, si db no es None, la función close() se llama en el objeto db para cerrar la conexión de la base de
        db.close()

#Creamos una función para añadir la configuración que cierra la conexión al terminar de realizar la petición.
def init_app(app):
    app.teardown_appcontext(close_db)
```

3. Añadimos la configuración de cierre de la conexión bbdd a nuestro archivo ‘__init__.py’

```
import os # (operative system) usamos esta libreria para acceder a las variables de entorno.
from flask import Flask #importamos flask.

#para trabajar con esta separacion de modulos debemos crear la funcion create_app.
def create_app(): #servirá para hacer testing o para crear instancias de la app.
    app = Flask(__name__) #Toda aplicación creada en Flask es una instancia de la clase Flask.

#Configuramos nuestra aplicación.
    app.config.from_mapping( #Vamos a definir variables de configuracion con from_mapping.
        SECRET_KEY = 'mikey', #Llave(cookie) a utilizar para definir las sesiones.
    #definimos el host, password, user y ddbb a partir de variables de entorno.
        DATABASE_HOST = os.environ.get('FLASK_DATABASE_HOST'),
        DATABASE_PASSWORD = os.environ.get('FLASK_DATABASE_PASSWORD'), #
        DATABASE_USER = os.environ.get('FLASK_DATABASE_USER'),
        DATABASE = os.environ.get('FLASK_DATABASE'),
    )

#Llamamos al archivo de bd.py para añadir la configuración de cierre de conexión.
    from . import db #El punto representa el directorio actual, y se utiliza para especificar la ruta relativa al archivo de origen
    db.init_app(app) #Desde el directorio actual estoy importando db.py y llamando a la función init_app.

#Creamos una ruta de pruebas.
    @app.route('/hola')
```

```
    def hola():
        return 'chanchito feliz'

    return app
```

4. De esta forma tenemos que nuestra aplicación (definida en '__init__.py') está llamando al servidor, se ejecuta la solicitud y entrega de datos, y se finaliza la conexión con la bbdd.

▼ Creando script de inicialización de la BBDD.

Vamos a escribir el script necesario para que podamos ejecutar un set de instrucciones que vamos a escribir en SQL. Este set de instrucciones que vamos a definir tiene como objetivo permitir que podamos crear las tablas que van a almacenar los datos de nuestra aplicación. Esto requiere que el script tenga acceso a la línea de comandos.

1. Dentro de db.py, vamos a crear la función init_db_command():

```
import mysql.connector #importamos el conector de mysql
import click #importamos click para poder para poder ejecutar comandos en la terminal
from flask import current_app, g #current_app mantiene la app que estamos ejecutando
                                 #g es una variable que está en toda nuestra app. La usaremos para almacenar el usuario.
from flask.cli import with_appcontext #con appcontext podemos acceder a las variables de la config.
from .schema import instructions #Lo creamos nosotros. Contiene los scripts para poder crear nuestra bbdd.

#Definimos una funcion que nos permita obtener nuestra ddbb y el cursor dentro de nuestra app.
def get_db():
    if 'db' not in g: #Si no se encuentra el atributo 'db' dentro de g
        g.db = mysql.connector.connect( #Vamos a generar una propiedad dentro de g que contendrá la conexión a la bbdd.
            host = current_app.config['DATABASE_HOST'], #Aquí usaremos la propiedad de current_app para definir las credenciales ir
            user = current_app.config['DATABASE_USER'],
            password = current_app.config['DATABASE_PASSWORD'],
            database = current_app.config['DATABASE'],
        )
        g.c = g.db.cursor(dictionary = True) #agregamos la obtención del cursor con acceso a las propiedades en forma de diccionari

    return g.db, g.c #Así al llamar a get_db() vamos a obtener la bbdd y el cursor.

#Definimos ahora una función que nos cierre la conexión de la bbdd cada vez que se realice una petición.
#La idea es que no debemos dejar esa conexión abierta con cada llamado.
def close_db(e = None):
    db = g.pop('db', None) #se encarga de obtener la variable db del objeto g que es un objeto global que se utiliza para almacenar
    #La función pop() busca la clave db en el diccionario g. Si se encuentra la clave db, se devuelve su valor y se elimina la clav
    if db is not None: #Luego, si db no es None, la función close() se llama en el objeto db para cerrar la conexión de la base de
        db.close()

#Creamos una función para añadir la configuración que cierra la conexión al terminar de realizar la petición.
def init_app(app):
    app.teardown_appcontext(close_db)

#Creamos la función que nos permitirá ejecutar instrucciones en la linea de comandos con objetivo de crear tablas.
def init_db(): # Esta función se encargará de la lógica.
    db, c = get_db() #Aquí llamamos a la base de datos y al cursor.

    #Las instrucciones que vamos a importar desde nuestro archivo schema deben estar en una lista y escritas en el orden correcto d
    for i in instructions: #iteramos por cada elemento de la lista instructions en schema.
        c.execute(i) #Ejecutando cada elemento de la lista como instrucción.
    db.commit()

@click.command('init-db') #Definimos el nombre que tendrá la instrucción en la línea de comandos para ejecutar esta función. Ej. "F
@with_appcontext #Esto indicará que se utilice el contexto de la aplicación para que pueda acceder a las variables de config en g.c

def init_db_command():
    init_db() #Esta funcion se hará cargo de la lógica para correr los scripts que definamos.
    click.echo('Base de datos inicializada') #para indicarnos que nuestro script ha acabado de correr con éxito.
```

▼ NOTA

Las instrucciones que vamos a importar desde nuestro archivo schema se van a encontrar todas escritas dentro de una lista, por lo que, en schema, las instrucciones deben estar escritas en el orden que quiero que se ejecuten y en una lista. Necesitamos esto porque la librería de MySQL Connector para conectarnos a MySQL, solo lee una instrucción a la vez. De esta forma, necesitamos usar una iteración de todas las instrucciones que vamos a ejecutar.

2. Creamos el archivo 'schema.py' donde definiremos las instrucciones SQL que ejecutará nuestra función 'init-db'

```
cd todo
touch schema.py
```

```
instructions = [ #Cada string será una instrucción/ elemento de la lista.
    'SET FOREING_KEY_CHECKS = 0;', #IMPORTANTE: Si queremos eliminar una tabla, no nos va a dejar si existen referencias de llaves
    'DROP TABLE IF EXISTS todo;', #Si la rabla existe, se elimina y se recrea.
    'DROP TABLE IF EXISTS user;', #Si la rabla existe, se elimina y se recrea.
    'SET FOREING_KEY_CHECKS = 1;', #Aquí volvemos a activar la validación de llaves foráneas referenciadas.
    #Con triple comilla doble se crea un string de multiples líneas.
    # -- Primero creo mi tabla de usuario.
    """
        CREATE TABLE user (
            id INT PRIMARY KEY AUTO_INCREMENT,
            username VARCHAR(50) UNIQUE NOT NULL,
            password VARCHAR(100) NOT NULL
        );
    """,
    # -- Ahora creo mi tabla de todo.
    """
        CREATE TABLE todo (
            id INT PRIMARY KEY AUTO_INCREMENT,
            created_by INT NOT NULL,
            created_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
            description TEXT NOT NULL,
            completed BOOLEAN NOT NULL,
            FOREING KEY (created_by) REFERENCES user (id)
        );
    """

]
```

▼ TABLA user

- id INT PRIMARY KEY AUTO_INCREMENT = Será un int, será llave primaria, y será autoincremental.
- username VARCHAR(50) UNIQUE NOT NULL = tendrá un máximo de 50 caracteres, será unico y no podrá ser nulo.
- password VARCHAR(100) NOT NULL = tendrá un máximo de 100 caracteres y no puede ser nulo.

▼ TABLA todo

- id INT PRIMARY KEY AUTO_INCREMENT = Será un int, será llave primaria, y será autoincremental.
- created_by INT NOT NULL = será un int no nulo. La idea es que se asigne una referencia directa a la tabla de usuario.
- created_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP = será una fecha, no nula, y se definirá por defecto al reloj/fecha actual del servidor MySQL.
- description TEXT NOT NULL = texto, no nulo.
- completed BOOLEAN NOT NULL = boleano (0, 1), no nulo. Otorga status de completado o no completado.
- FOREING KEY (created_by) REFERENCES user (id) = aquí hacemos la referencia de created_by al id en la tabla user.

▼ Ejecutamos el script

1. Definimos las variables de usuario en la línea de comandos. Para hacer esto detenemos la ejecución del programa.

```
export FLASK_DATABASE_HOST='localhost'
export FLASK_DATABASE_PASSWORD='Lalolanda123'
export FLASK_DATABASE_USER='dsc'
export FLASK_DATABASE='prueba'
```

2. Incluimos el comando init_db_command a la aplicación. Para esto, en ‘db.py’ incluiremos en el comando en la función init_app de la siguiente forma:

```
import mysql.connector #importamos el conector de mysql
import click #importamos click para poder para poder ejecutar comandos en la terminal
from flask import current_app, g #current_app mantiene la app que estamos ejecutando
                                 #g es una variable que está en toda nuestra app. La usaremos para almacenar el usuario.
from flask.cli import with_appcontext #con appcontext podemos acceder a las variables de la config.
from .schema import instructions #Lo creamos nosotros. Contiene los scripts para poder crear nuestra bbdd.

#Definimos una funcion que nos permita obtener nuestra ddbb y el cursor dentro de nuestra app.
def get_db():
    if 'db' not in g: #Si no se encuentra el atributo 'db' dentro de g
        g.db = mysql.connector.connect( #Vamos a generar una propiedad dentro de g que contendrá la conexión a la bbdd.
            host = current_app.config['DATABASE_HOST'], #Aquí usaremos la propiedad de current_app para definir las credenciales ingres
            user = current_app.config['DATABASE_USER'],
            password = current_app.config['DATABASE_PASSWORD'],
            database = current_app.config['DATABASE'],
        )
        g.c = g.db.cursor(dictionary = True) #agregamos la obtención del cursor con acceso a las propiedades en forma de diccionario

    return g.db, g.c #Así al llamar a get_db() vamos a obtener la bbdd y el cursor.

#Definimos ahora una función que nos cierre la conexión de la bbdd cada vez que se realice una petición.
#La idea es que no debemos dejar esa conexión abierta con cada llamado.
def close_db(e = None):
    db = g.pop('db', None) #se encarga de obtener la variable db del objeto g que es un objeto global que se utiliza para almacenar var
    #La función pop() busca la clave db en el diccionario g. Si se encuentra la clave db, se devuelve su valor y se elimina la clave db
    if db is not None: #Luego, si db no es None, la función close() se llama en el objeto db para cerrar la conexión de la base de dato
        db.close()

#Creamos una función para añadir la configuración que cierra la conexión al terminar de realizar la petición.
def init_app(app):
    app.teardown_appcontext(close_db)
    app.cli.add_command(init_db_command)

#Creamos la función que nos permitirá ejecutar instrucciones en la linea de comandos con objetivo de crear tablas.
def init_db(): # Esta función se encargará de la lógica.
    db, c = get_db() #Aquí llamamos a la base de datos y al cursor.

    #Las instrucciones que vamos a importar desde nuestro archivo schema deben estar en una lista y escritas en el orden correcto de ej
    for i in instructions: #iteramos por cada elemento de la lista instructions en schema.
        c.execute(i) #Ejecutando cada elemento de la lista como instrucción.
    db.commit()

@click.command('init-db') #Definimos el nombre que tendrá la instrucción en la línea de comandos para ejecutar esta función. Ej. "Flask
@with_appcontext #Esto indicará que se utilice el contexto de la aplicación para que pueda acceder a las variables de config en g.db.

def init_db_command():
    init_db() #Esta funcion se hará cargo de la lógica para correr los scripts que definamos.
    click.echo('Base de datos inicializada') #para indicarnos que nuestro script ha acabado de correr con éxito.
```

Esto suscribirá el comando a la aplicación, permitiendonos ejecutar el comando

```
flask init-db
```

▼ Modulo de autenticación

*En Flask los modulos se agrupan como un **blueprint**. Por ejemplo, si vamos a tener un blueprint de autenticación, este debería incorporar los modulos de inicio de sesión, registro, autenticación y firewall.*

1. Vamos a crear un nuevo archivo llamado ‘auth.py’

   ```
   touch auth.py
   ```

   Dentro de auth:

   ```
   import functools #set de funciones para construir aplicaciones.
   from flask import(
       Blueprint, #Nos permite crear blueprints configurables.
       flash, #Funcion que nos permite enviar mensaje de forma generica a nuestras plantillas.
       g, #variable trascendental a la apricación.
       render_template, #renderiza plantillas
   ```

```
    request, #vamos a recibir datos desde un formulario.
    url_for, #vamos a crear urls.
    session, #para poder mantener una referencia del usuario que se encuentra en el contexto actual interactuando con la app.
    redirect
)
from werkzeug.security import (
    check_password_hash, #verifica que la contraseña que se ingresa sea igual a otra.
    generate_password_hash #encripta la contraseña que se está enviando.
)

from todo.db import get_db #importamos esto para poder interactuar con la bbdd.
```

2. Creamos el Blueprint “bp”:

```
import functools #set de funciones para construir aplicaciones.
from flask import(
    Blueprint, #Nos permite crear blueprints configurables.
    flash, #Funcion que nos permite enviar mensaje de forma generica a nuestras plantillas.
    g, #variable trascendental a la apricación.
    render_template, #renderiza plantillas
    request, #vamos a recibir datos desde un formulario.
    url_for, #vamos a crear urls.
    session, #para poder mantener una referencia del usuario que se encuentra en el contexto actual interactuando con la app.
    redirect
)
from werkzeug.security import (
    check_password_hash, #verifica que la contraseña que se ingresa sea igual a otra.
    generate_password_hash #encripta la contraseña que se está enviando.
)

from todo.db import get_db #importamos esto para poder interactuar con la bbdd.

bp = Blueprint('auth', __name__, url_prefix='/auth') #Es importante registrar este blueprint en la aplicación (__init__.py).
```

3. Registramos este nuevo Blueprint en la aplicación (’__init__.py’)

```
import os # (operative system) usamos esta libreria para acceder a las variables de entorno.
from flask import Flask #importamos flask.

#para trabajar con esta separacion de modulos debemos crear la funcion create_app.
def create_app(): #servirá para hacer testing o para crear instancias de la app.
    app = Flask(__name__) #Toda aplicación creada en Flask es una instancia de la clase Flask.

#Configuramos nuestra aplicación.
    app.config.from_mapping( #Vamos a definir variables de configuracion con from_mapping.
        SECRET_KEY = 'mikey', #Llave(cookie) a utilizar para definir las sesiones.
    #definimos el host, password, user y ddbb a partir de variables de entorno.
        DATABASE_HOST = os.environ.get('FLASK_DATABASE_HOST'),
        DATABASE_PASSWORD = os.environ.get('FLASK_DATABASE_PASSWORD'), #
        DATABASE_USER = os.environ.get('FLASK_DATABASE_USER'),
        DATABASE = os.environ.get('FLASK_DATABASE'),
    )

#Llamamos al archivo de bd.py para añadir la configuración de cierre de conexión.
    from . import db #El punto representa el directorio actual, y se utiliza para especificar la ruta relativa al archivo de origen
    db.init_app(app) #Desde el directorio actual estoy importando db.py y llamando a la función init_app.

#Inscribimos el blueprint creado en auth.py en la aplicación.
    from . import auth
    app.register_blueprint(auth.bp) #bp es el nombre que le dimos a la variable blueprint creada en auth.py

#Creamos una ruta de pruebas.
    @app.route('/hola')
    def hola():
        return 'chanchito feliz'

    return app
```

4. Creamos una nueva ruta para nuestro blueprint con la función de registro en el archivo ‘auth.py’:

```
import functools #set de funciones para construir aplicaciones.
from flask import(
```

```
    Blueprint, #Nos permite crear blueprints configurables.
    flash, #Funcion que nos permite enviar mensaje de forma generica a nuestras plantillas.
    g, #variable trascendental a la apricación.
    render_template, #renderiza plantillas
    request, #vamos a recibir datos desde un formulario.
    url_for, #vamos a crear urls.
    session, #para poder mantener una referencia del usuario que se encuentra en el contexto actual interactuando con la app.
    redirect
)
from werkzeug.security import (
    check_password_hash, #verifica que la contraseña que se ingresa sea igual a otra.
    generate_password_hash #encripta la contraseña que se está enviando.
)

from todo.db import get_db #importamos esto para poder interactuar con la bbdd.

bp = Blueprint('auth', __name__, url_prefix='/auth') #Es importante registrar este blueprint en la aplicación (__init__.py).

#Agregamos una ruta dentro de nuestro blueprint
@bp.route('/register', methods=['GET', 'POST']) #ruta /register, metodos que manejará la ruta.
def register():
    if request.method == 'POST': #Primero validamos dentro del request que se está enviando al sv sea el de POST.
        username = request.form['username'] #Vamos a obtener username.
        password = request.form['password'] #Vamos a obtener password.
        db, c = get_db() #Vamos a buscar la base y el cursor.
        error = None #definimos una variable de error para mas adelante usarla como flash.
        c.execute( #Ejecutamos la siguiente consulta a nuestra bbdd
            'select id from user where username = %s', (username,) #Esta consulta selecciona el id si el username en user es distin
        )
# -- Definimos los 3 errores posibles para el registro de un usuario con 2 campos.
    # 1. Que no se registre un username.
        if not username: #Si no tenemos un username de parte del usuario entregamos un mensaje de error.
            error = 'Username es requerido'
    # 2. Que no se registre una password.
        if not password: #Si no tenemos un password de parte del usuario entregamos un mensaje de error.
            error = 'Password es requerido'
    # 3. Que el username registrado ya exista.
        elif c.fetchone() is not None: #Si, por el contrario, tenemos un username, vamos a buscar si existe la coincidencia dentro
            error = 'Usuario {} ya se encuentra registrado.'.format(username) #Esta sintaix con {} es solo posible si se utiliza el
# -- Validamos que el error no exista para ejecutar el registro.
        if error is None:
            c.execute(
                'insert into user (username, password) values (%s, %s)',
                (username, generate_password_hash(password)) #aquí llamamos el username y la contraseña encriptada con generate_pas
            )
            db.commit() #commit para comprometer la bbdd.

            return redirect(url_for('auth.login')) #Una vez que se haya completado el registro enviamos al usuario a auth.login (de

        flash(error) #Si no tuvimos un caso de éxito enviamos mensaje de error.

    return render_template('auth/register.html') #retornamos en caso de que la petición no sea POST sino GET.
```

5. Ahora creamos una función de login en nuestro blueprint:

```
import functools #set de funciones para construir aplicaciones.
from flask import(
    Blueprint, #Nos permite crear blueprints configurables.
    flash, #Funcion que nos permite enviar mensaje de forma generica a nuestras plantillas.
    g, #variable trascendental a la apricación.
    render_template, #renderiza plantillas
    request, #vamos a recibir datos desde un formulario.
    url_for, #vamos a crear urls.
    session, #para poder mantener una referencia del usuario que se encuentra en el contexto actual interactuando con la app.
    redirect
)
from werkzeug.security import (
    check_password_hash, #verifica que la contraseña que se ingresa sea igual a otra.
    generate_password_hash #encripta la contraseña que se está enviando.
)

from todo.db import get_db #importamos esto para poder interactuar con la bbdd.

bp = Blueprint('auth', __name__, url_prefix='/auth') #Es importante registrar este blueprint en la aplicación (__init__.py).

#Agregamos una ruta de registro dentro de nuestro blueprint
@bp.route('/register', methods=['GET', 'POST']) #ruta /register, metodos que manejará la ruta.
def register():
```

```
    if request.method == 'POST': #Primero validamos dentro del request que se está enviando al sv sea el de POST.
        username = request.form['username'] #Vamos a obtener username.
        password = request.form['password'] #Vamos a obtener password.
        db, c = get_db() #Vamos a buscar la base y el cursor.
        error = None #definimos una variable de error para mas adelante usarla como flash.
        c.execute( #Ejecutamos la siguiente consulta a nuestra bbdd
            'select id from user where username = %s' #Esta consulta selecciona el id si el username en user es distinto de None.
        )
# -- Definimos los 3 errores posibles para el registro de un usuario con 2 campos.
    # 1. Que no se registre un username.
        if not username: #Si no tenemos un username de parte del usuario entregamos un mensaje de error.
            error = 'Username es requerido'
    # 2. Que no se registre una password.
        if not password: #Si no tenemos un password de parte del usuario entregamos un mensaje de error.
            error = 'Password es requerido'
    # 3. Que el username registrado ya exista.
        elif c.fetchone() is not None: #Si, por el contrario, tenemos un username, vamos a buscar si existe la coincidencia dentro
            error = 'Usuario {} ya se encuentra registrado.'.format(username) #Esta sintaix con {} es solo posible si se utiliza el
# -- Validamos que el error no exista para ejecutar el registro.
        if error is None:
            c.execute(
                'insert into user (username, password) values (%s, %s)',
                (username, generate_password_hash(password)) #aquí llamamos el username y la contraseña encriptada con generate_pas
            )
            db.commit() #commit para comprometer la bbdd.

            return redirect(url_for('auth.login')) #Una vez que se haya completado el registro enviamos al usuario a auth.login (de

        flash(error) #Si no tuvimos un caso de éxito enviamos mensaje de error.

    return render_template('auth/register.html') #retornamos en caso de que la petición no sea POST sino GET.

#Agregamos una ruta de login dentro de nuestro blueprint
@bp.route('/login', methods=['GET', 'POST']) #ruta /register, metodos que manejará la ruta.
def login():
    if request.method == 'POST': #Primero validamos dentro del request que se está enviando al sv sea el de POST.
        username = request.form['username'] #Vamos a obtener username.
        password = request.form['password'] #Vamos a obtener password.
        db, c = get_db() #Vamos a buscar la base y el cursor.
        error = None #definimos una variable de error para mas adelante usarla como flash.
        c.execute( #Ejecutamos la siguiente consulta a nuestra bbdd
            'select * from user where username = %s', (username,) #buscamos si existe algun usuario con un username coincidente cor
        )
        user = c.fetchone() #definimos la variable user como el registro coincidente de la bbdd con username.
# -- Verificamos errores
    # 1. Si no existe coincidencia en la bbdd.
        if user is None:
            error = 'Usuario y/o contraseña inválida' #Mensaje ambiguo para evitar obtener datos por la fuerza.
    # 2. Si existe coincidencia de usuario pero no de contraseña.
        elif not check_password_hash(user['password'], password):
            error = 'Usuario y/o contraseña inválida' #Mismo mensaje ambiguo para evitar entregar información específica respecto a

# -- En caso de que no exista error
        if error is None:
            session.clear() #Vamos a limpiar la sesión.
            session['user_id'] = user['id'] #creamos variable dentro de la sesión que vamos a llamar user_id, y le pasaremos el id
            return redirect(url_for('todo.index')) #redirigimos al usuario a la página de inicio.

        flash(error) # Caso en el que el usuario tuvo errores.

    return render_template('auth/login.html') #retornamos en caso de que la petición no sea POST sino GET.
```

▼ Creamos nuestro HTML base

1. Primero creamos la carpeta ‘templates’ dentro de la carpeta ‘todo’, y dentro de ‘templates’ agregamos el archivo ‘base.html’.

```
cd todo
mkdir templates
cd templates
touch base.html
```

2. Editamos el archivo HTML.

```
<!DOCTYPE html>
<title>
    {% block title %}{% endblock%} - Todo List <!--Bloque de titulo.-->
</title>
<link rel="stylesheet" href="{{url_for('static', filename='style.css')}}"><!--Aquí incluimos nuestro archivo CSS-->
<nav><!--Aquí escribiremos una estructura HTML con una pequeña interfaz de usuario-->
    <h1>Todo List</h1>
    <ul>
        {% if g.user %}<!--Pregunto si dentro de la variable g existe user-->
            <li><span>{{ g.user['username']}}</span></li><!--En caso de existir vamos a buscar el username-->
            <li><a href="{{ url_for('auth.logout')}}">Cerrar Sesión</a></li><!--Le daremos un link para el cierre de sesión-->
        {% else %}
            <li><a href="{{ url_for('auth.register')}}">Registrarse</a></li><!--Link a la ruta de registro-->
            <li><a href="{{ url_for('auth.login')}}">Iniciar Sesión</a></li><!-- Link a la ruta de inicio de sesión-->
        {% endif %}
    </ul>
</nav>

<!-- SECCIÓN PARA MENSAJES FLASH -->
    <!-- Aquí desplegaremos los mensajes flash definidos en auth.py-->
<section class="content">
    <header>
        {% block header %}{% endblock %}
    </header>
    {% for message in get_flashed_messages() %}
    <div class="flash">{{ message }}</div>
    {% endfor %}
    {% block content %}{% endblock %}<!-- Bloque para desplegar contenido -->
</section>
```

▼ NOTA

La sintaxis `{% ... %}` se utiliza para indicar que se está utilizando una etiqueta de plantilla en lugar de una etiqueta HTML normal. Estas etiquetas de plantilla pueden incluir expresiones, declaraciones condicionales, ciclos, bloques de contenido y más. En resumen, los bloques de contenido como `{% block ... %}{% endblock %}` son una herramienta poderosa para crear plantillas dinámicas y reutilizables en aplicaciones web.

▼ Creamos plantillas de Registro y Login

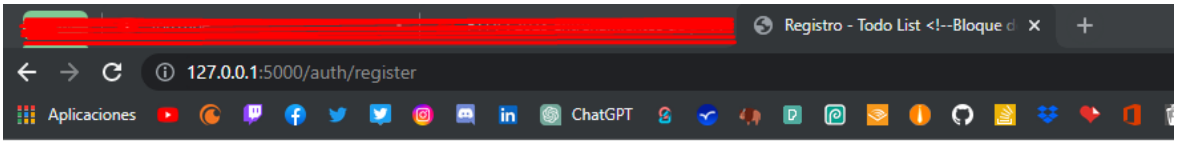
1. Vamos a crear una carpeta de auth y dentro de esta crearemos los archivos html para registro y login. Haremos esto porque dentro del archivo ‘auth.py’ definimos que estas plantillas se encontrarían dentro de una carpeta llamada auth, en una ruta similar a ‘auth/login.html’.

```
cd templates
mkdir auth
cd auth
touch register.html
touch login.html
```

2. Editamos el archivo de Registro

```
{% extends 'base.html'%}<!--Extendemos la plantilla desde base.html-->
{% block header %}
    <h1>{% block title %}Registro{% endblock %}</h1>
{% endblock %}

{% block content %}
    <form method="post"><!--Agregamos un formulario para el registro-->
        <label for="username">Username</label><!--Etiqueta para Username-->
        <input name="username" id="username" required /><!--definimos el input como "name", pues es el nombre de la variable con la
        <label for="password">Password</label><!--Etiqueta para Password-->
        <input type="password" name="password" id="password" required /><!--importante que el input es tipo password para que nos m
        <input type="submit" value="Registrarse" /><!--Boton para enviar el formulario de este endpoint de registro-->
    </form>
{% endblock %}
```



Todo List

- Registrarse
- Iniciar Sesión

Registro

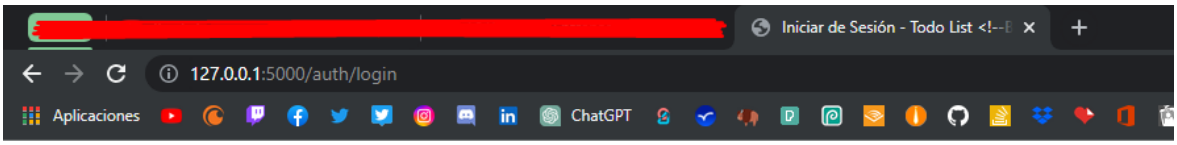
Username Password Registrarse

3. Editamos el archivo de Inicio de Sesión

```
{% extends 'base.html' %}<!--Extendemos la plantilla desde base.html-->
{% block header %}
    <h1>{% block title %}Iniciar de Sesión{% endblock %}</h1>
{% endblock %}

{% block content %}
    <form method="post">
        <label for="username">Username</label>
        <input name="username" id="username" required />
        <label for="password">Password</label>
        <input type="password" name="password" id="password" required />
        <input type="submit" value="Iniciar Sesión" />

    </form>
{% endblock %}
```



Todo List

- Registrarse
- Iniciar Sesión

Iniciar de Sesión

Username Password Iniciar Sesión

▼ Blueprint ToDo e ‘index.html’

1. Vamos a crear un nuevo blueprint llamado ‘todo.py’ en la carpeta todo.

```
cd ..
touch todo.py
```

2. Editamos ‘todo.py’ para crear el blueprint

```
from flask import (
    Blueprint, flash, g, redirect, render_template, request, url_for
)
from werkzeug.exceptions import abort #En caso de que un usuario intente actualizar un ToDo que no le pertenezca.
# -- Vamos a crear esta función para proteger el endpoint solicitando siempre que exista la sesión del usuario.
from todo.auth import login_required #Crearemos la función apenas terminemos de registrar el blueprint.
from todo.db import get_db

bp = Blueprint('todo', __name__) #Creamos el blueprint de nombre 'todo'.
```

3. Registramos el nuevo blueprint en ‘__init__.py’.

```
import os # (operative system) usamos esta libreria para acceder a las variables de entorno.
from flask import Flask #importamos flask.

#para trabajar con esta separacion de modulos debemos crear la funcion create_app.
def create_app(): #servirá para hacer testing o para crear instancias de la app.
    app = Flask(__name__) #Toda aplicación creada en Flask es una instancia de la clase Flask.

#Configuramos nuestra aplicación.
    app.config.from_mapping( #Vamos a definir variables de configuracion con from_mapping.
        SECRET_KEY = 'mikey', #Llave(cookie) a utilizar para definir las sesiones.
    #definimos el host, password, user y ddbb a partir de variables de entorno.
        DATABASE_HOST = os.environ.get('FLASK_DATABASE_HOST'),
        DATABASE_PASSWORD = os.environ.get('FLASK_DATABASE_PASSWORD'), #
        DATABASE_USER = os.environ.get('FLASK_DATABASE_USER'),
        DATABASE = os.environ.get('FLASK_DATABASE'),
    )

#Llamamos al archivo de bd.py para añadir la configuración de cierre de conexión.
    from . import db #El punto representa el directorio actual, y se utiliza para especificar la ruta relativa al archivo de origen
    db.init_app(app) #Desde el directorio actual estoy importando db.py y llamando a la función init_app.

#Inscribimos el blueprint creado en auth.py en la aplicación.
    from . import auth
    app.register_blueprint(auth.bp) #bp es el nombre que le dimos a la variable blueprint creada en auth.py

#Inscribimos el blueprint de todo.
    from . import todo
    app.register_blueprint(todo.bp)

#Creamos una ruta de pruebas.
    @app.route('/hola')
    def hola():
        return 'chanchito feliz'

    return app
```

4. Creamos una función para proteger nuestras rutas en ‘auth.py’.

   Definimos que haríamos esto en el paso 2 con la función ‘login_required’. Haremos esto para proteger el uso del endpoint solicitando que siempre exista un inicio de sesión activo para generar edición del ToDo.

   La función que vamos a definir es una función decoradora. Lo que hace esta función es recibir como argumento la función que queremos decorar.

```
import functools #set de funciones para construir aplicaciones.
from flask import(
    Blueprint, #Nos permite crear blueprints configurables.
    flash, #Funcion que nos permite enviar mensaje de forma generica a nuestras plantillas.
    g, #variable trascendental a la apricación.
    render_template, #renderiza plantillas
    request, #vamos a recibir datos desde un formulario.
    url_for, #vamos a crear urls.
    session, #para poder mantener una referencia del usuario que se encuentra en el contexto actual interactuando con la app.
```

```
    redirect
)
from werkzeug.security import (
    check_password_hash, #verifica que la contraseña que se ingresa sea igual a otra.
    generate_password_hash #encripta la contraseña que se está enviando.
)

from todo.db import get_db #importamos esto para poder interactuar con la bbdd.

bp = Blueprint('auth', __name__, url_prefix='/auth') #Es importante registrar este blueprint en la aplicación (__init__.py).

### - Agregamos una ruta de registro dentro de nuestro blueprint
@bp.route('/register', methods=['GET', 'POST']) #ruta /register, metodos que manejará la ruta.
def register():
    if request.method == 'POST': #Primero validamos dentro del request que se está enviando al sv sea el de POST.
        username = request.form['username'] #Vamos a obtener username.
        password = request.form['password'] #Vamos a obtener password.
        db, c = get_db() #Vamos a buscar la base y el cursor.
        error = None #definimos una variable de error para mas adelante usarla como flash.
        c.execute( #Ejecutamos la siguiente consulta a nuestra bbdd
            'select id from user where username = %s', #Esta consulta selecciona el id si el username en user es distinto de None.
            (username,) #Importante agregar el segundo argumento.
        )
# -- Definimos los 3 errores posibles para el registro de un usuario con 2 campos.
    # 1. Que no se registre un username.
        if not username: #Si no tenemos un username de parte del usuario entregamos un mensaje de error.
            error = 'Username es requerido'
    # 2. Que no se registre una password.
        if not password: #Si no tenemos un password de parte del usuario entregamos un mensaje de error.
            error = 'Password es requerido'
    # 3. Que el username registrado ya exista.
        elif c.fetchone() is not None: #Si, por el contrario, tenemos un username, vamos a buscar si existe la coincidencia dentro
            error = 'Usuario {} ya se encuentra registrado.'.format(username) #Esta sintaix con {} es solo posible si se utiliza el
# -- Validamos que el error no exista para ejecutar el registro.
        if error is None:
            c.execute(
                'insert into user (username, password) values (%s, %s)',
                (username, generate_password_hash(password)) #aquí llamamos el username y la contraseña encriptada con generate_pas
            )
            db.commit() #commit para comprometer la bbdd.

            return redirect(url_for('auth.login')) #Una vez que se haya completado el registro enviamos al usuario a auth.login (de

        flash(error) #Si no tuvimos un caso de éxito enviamos mensaje de error.

    return render_template('auth/register.html') #retornamos en caso de que la petición no sea POST sino GET.

### - Agregamos una ruta de login dentro de nuestro blueprint
@bp.route('/login', methods=['GET', 'POST']) #ruta /register, metodos que manejará la ruta.
def login():
    if request.method == 'POST': #Primero validamos dentro del request que se está enviando al sv sea el de POST.
        username = request.form['username'] #Vamos a obtener username.
        password = request.form['password'] #Vamos a obtener password.
        db, c = get_db() #Vamos a buscar la base y el cursor.
        error = None #definimos una variable de error para mas adelante usarla como flash.
        c.execute( #Ejecutamos la siguiente consulta a nuestra bbdd
            'select * from user where username = %s', (username,) #buscamos si existe algun usuario con un username coincidente cor
        )
        user = c.fetchone() #definimos la variable user como el registro coincidente de la bbdd con username.
# -- Verificamos errores
    # 1. Si no existe coincidencia en la bbdd.
        if user is None:
            error = 'Usuario y/o contraseña inválida' #Mensaje ambiguo para evitar obtener datos por la fuerza.
    # 2. Si existe coincidencia de usuario pero no de contraseña.
        elif not check_password_hash(user['password'], password):
            error = 'Usuario y/o contraseña inválida' #Mismo mensaje ambiguo para evitar entregar información específica respecto a

# -- En caso de que no exista error
        if error is None:
            session.clear() #Vamos a limpiar la sesión.
            session['user_id'] = user['id'] #creamos variable dentro de la sesión que vamos a llamar user_id, y le pasaremos el id
            return redirect(url_for('todo.index')) #redirigimos al usuario a la página de inicio.

        flash(error) # Caso en el que el usuario tuvo errores.

    return render_template('auth/login.html') #retornamos en caso de que la petición no sea POST sino GET.

### - FUNCION DECORADORA PARA CHEQUEAR EL LOGIN.
    # 1. Primero creamos una función que asigne el usuario dentro de la variable trascendental 'g'.
@bp.before_app_request
def load_logged_in_user(): #definimos la función.
```

```
    user_id = session.get('user_id') #vamos a sacar el user_id dentro de nuestro objeto de sesión.

    if user_id is None: #comprobamos si se obtuvo algo desde el objeto de sesión.
        g.user = None #si no se obtuvo nada, dejamos g.user como None.
    else:
        db, c = get_db() #Si existe, vamos a ir a buscar con el user_id el usuario a la bbdd.
        c.execute(
            'select * from user where id = %s', (user_id,) #busca en base al id que guardamos en sesión el registro en la bbdd.
        )
        g.user = c.fetchone() # Guarda en g.user el primer elemento.

    # 2. Definimos la función que protegerá los ToDos requiriendo el login.
def login_required(view):
    @functools.wraps(view) #Envolvemos a la función utilizando la función de wraps.
    def wrapped_view(**kwargs): #
        if g.user is None: #Preguntamos si g.user es None. Si es afirmativo, el usuario no ha iniciado sesión.
            return redirect(url_for('auth.login')) #Redireccionamos al usuario a la pagina de login.
        return view(**kwargs) #Si el usuario inició sesión, vamos a devolver la vista y los argumentos.
    return wrapped_view #Devolvemos la función recién creada.
```

▼ Explicación de la función login_required:

La función `login_required` es un decorador en Python que se utiliza para proteger una vista o función de acceso no autorizado. La función toma una vista como argumento y devuelve una nueva vista que comprueba si el usuario ha iniciado sesión antes de llamar a la vista original.

La función envuelve la vista original en una nueva vista que comprueba si `g.user` es nulo (es decir, si el usuario no ha iniciado sesión). Si `g.user` es nulo, la función redirige al usuario a la página de inicio de sesión utilizando la función `redirect` de Flask, que devuelve una respuesta HTTP 302 que redirige al usuario a la URL especificada. Si `g.user` no es nulo, la función llama a la vista original con los argumentos correspondientes (**kwargs).

La función `functools.wraps` se utiliza para conservar el nombre de la función y la documentación de la función original en la nueva función envuelta. Esto es útil porque las funciones decoradas pueden ser difíciles de depurar si pierden su nombre o documentación original.

En resumen, la función `login_required` es un decorador que protege una vista o función de acceso no autorizado verificando si el usuario ha iniciado sesión. Si el usuario no ha iniciado sesión, se redirige al usuario a la página de inicio de sesión. Si el usuario ha iniciado sesión, la vista original se llama con los argumentos correspondientes.

5. Definimos ahora la función de index en ‘todo.py’:

   La función index listará todos los ToDo que pertenezcan al usuario.

```
from flask import (
    Blueprint, flash, g, redirect, render_template, request, url_for
)
from werkzeug.exceptions import abort #En caso de que un usuario intente actualizar un ToDo que no le pertenezca.
# -- Vamos a crear esta función para proteger el endpoint solicitando siempre que exista la sesión del usuario.
from todo.auth import login_required #Crearemos la función apenas terminemos de registrar el blueprint.
from todo.db import get_db

bp = Blueprint('todo', __name__) #Creamos el blueprint de nombre 'todo'.

@bp.route('/') #Este decorador se utiliza para indicar la URL que se debe asociar con la vista. En este caso, la URL es simplemente
@login_required #Una vez se ingrese en esta ruta, se chequea el inicio de sesión
def index(): #si el inicio de sesión es correcto, se activa la función index que listará los todo del usuario.
    db, c = get_db()
    c.execute(
        'select t.id, t.description, u.username, t.completed, t.created_at from todo t JOIN user u on t.created_by = u.id order by
        #Los elementos consultados (t.id, t.description, u.username, t.completed, t.created_at)
        #deben cumplir con que el created_by en la tabla de todo, sea igual al id en la tabla de user
        #y vendrán ordenados en base a created_at y de forma descendente.
    )
    todos = c.fetchall()
    return render_template('todo/index.html', todos = todos) #renderizar una plantilla HTML llamada "index.html" y enviarla como re
    #La plantilla "index.html" es renderizada con los valores pasados a través del diccionario todos = todos. El diccionario todos
```

6. Creamos el archivo index.html dentro templates en una nueva carpeta llamada todo (templates/todo/index.html).

```
cd templates
mkdir todo
cd todo
touch index.html
```

7. Editamos el archivo ‘index.html’

```
{% extends 'base.html' %} <!--Extendemos la plantilla desde base.html-->
{% block header %}
    <h1>{% block title %}ToDos{% endblock %}</h1>
        {% if g.user %} <!--Si el usuario se encuentra correctamente-->
            <a class="action" href="{{ url_for('todo.create') }}">Nuevo</a><!--Vamos a darle la opción de crear un nuevo todo (vamo
        {% endif %}
{% endblock %}

{% block content %}
<ul>
    {% for todo in todos %}<!--Con esta iteración vamos a imprimir todos los ToDos del usuario-->
    <li class="todo">
        <div>
            <h1>
                {% if todo['completed'] == 1 %}<!--Si el ToDo se encuentra completado-->
                <strike>{{ todo['desctription']}}</strike><!--Imprime el ToDo con una raya sobre el texto(para esto sirve strike).-
                {% else %}<!--Si no se encuentra completado-->
                {{ todo['desctription'] }}<!--Se imprime solo el ToDo-->
                {% endif %}
            </h1>
            <div class="about">{{ todo['created_at'].strftime('%Y-%m-%d') }}</div><!--Mostramos la fecha de creación del ToDo-->
            <!--Generamos un botón para que el usuario pueda editar un ToDo. El id del ToDo a actualizar será otorgado a través de
            <a class="action" href="{{ url_for('todo.update', id=todo['id']) }}">Editar</a>
        </div>
    </li>
    {% if not loop.last %}<!--Si el ToDo no es el último-->
        <hr><!--Mostramos esta etiqueta de hr-->
    {% endif %}
    {% endfor %}
</ul>
{% endblock%}
```

▼ ¿Que hace la etiqueta de hr?

La etiqueta HTML `<hr>` se utiliza para crear una línea horizontal en una página web. Esta etiqueta representa un "quiebre temático" y se utiliza para separar visualmente el contenido de la página en secciones distintas.

La etiqueta `<hr>` es una etiqueta de cierre automático, lo que significa que no necesita una etiqueta de cierre correspondiente. Puede incluir atributos opcionales, como `color`, `size` y `width`, para personalizar la apariencia de la línea.

Por ejemplo, el siguiente código HTML creará una línea horizontal en una página web:

```
<h2>Sección 1</h2>
<p>Este es el contenido de la sección 1.</p>
<hr>
<h2>Sección 2</h2>
<p>Este es el contenido de la sección 2.</p>
```

En este caso, la etiqueta `<hr>` se utiliza para separar visualmente las secciones 1 y 2 del contenido de la página. La línea horizontal creada por la etiqueta `<hr>` puede ser personalizada utilizando atributos opcionales, como se mencionó anteriormente.

▼ Creamos las funciones de ‘todo.create’ y ‘todo.update’ en ‘todo.py’.

1. Escribimos la ruta de todo.create en ‘todo.py’:

```
from flask import (
    Blueprint, flash, g, redirect, render_template, request, url_for
)
from werkzeug.exceptions import abort #En caso de que un usuario intente actualizar un ToDo que no le pertenezca.
```

```
# -- Vamos a crear esta función para proteger el endpoint solicitando siempre que exista la sesión del usuario.
from todo.auth import login_required #Crearemos la función apenas terminemos de registrar el blueprint.
from todo.db import get_db

bp = Blueprint('todo', __name__) #Creamos el blueprint de nombre 'todo'.

@bp.route('/') #Este decorador se utiliza para indicar la URL que se debe asociar con la vista. En este caso, la URL es simplemente
@login_required #Una vez se ingrese en esta ruta, se chequea el inicio de sesión
def index(): #si el inicio de sesión es correcto, se activa la función index que listará los todo del usuario.
    db, c = get_db()
    c.execute(
        'select t.id, t.description, u.username, t.completed, t.created_at from todo t JOIN user u on t.created_by = u.id order by
        #Los elementos consultados (t.id, t.description, u.username, t.completed, t.created_at)
        #deben cumplir con que el created_by en la tabla de todo, sea igual al id en la tabla de user
        #y vendrán ordenados en base a created_at y de forma descendente.
    )
    todos = c.fetchall()
    return render_template('todo/index.html', todos = todos) #renderizar una plantilla HTML llamada "index.html" y enviarla como re
    #La plantilla "index.html" es renderizada con los valores pasados a través del diccionario todos = todos. El diccionario todos

@bp.route('/create', methods=['GET', 'POST']) #Creamos la ruta y asignamos los métodos.
@login_required #solicitamos que se cumpla login_required.
def create(): #definimos la función create.
    if request.method == 'POST': #verificamos que se esté usando metodo post
        description =  request.form['description'] # Tomamos la descripción desde request.form
        error = None #creamos un mensaje de error inofensivo.

        if not description: #Si la descripción no se ingresó
            error = 'Descripción es requerida' #Le indicamos al usuario el error.
        if error is not None: #Verificamos un cambio en error.
            flash(error) #Flasheamos un error
        else: #Si no hay error finalmente procedemos.
            db, c = get_db() # Obtenemos la bbdd y el cursor
            c.execute(
                #Ejecutamos un codigo SQL en la bbdd para insertar datos en la bbdd "todo".
                'insert into todo (description, completed, created_by)'
                'values (%s, %s, %s)',
                #Insertamos description como el input del usuario, false para indicar que la completitud sea 0, y g.user['id'] para
                (description, False, g.user['id'])
            )
            db.commit() #Comprometemos la bbdd.
            return redirect(url_for('todo.index')) #Redireccionamos al usuario a su pagina principal donde se muestran los ToDo (in
    return render_template('todo/create.html') #Asignamos la renderización de una plantilla en create.html
```

2. Creamos el archivo html y editamos.

```
cd templates/todo
touch create.html
```

```
{% extends 'base.html' %}<!--Extendemos la plantilla desde base.html-->
{% block header %}
    <h1>{% block title %}Nuevo ToDo{% endblock %}</h1>
{% endblock %}

{% block content %}
    <form method="post">
        <label for="description">Descripción</label><!--Solicitamos la descripción del ToDo-->
        <input name="description" id="description" required /><!--Entregamos la caja para el input-->
        <input type="submit" value="Guardar" /><!--Entregamos un boton para guardar-->
    </form>
{% endblock%}
```

3. Escribimos la ruta del todo.update en ‘todo.py’

```
from flask import (
    Blueprint, flash, g, redirect, render_template, request, url_for
)
from werkzeug.exceptions import abort #En caso de que un usuario intente actualizar un ToDo que no le pertenezca.
# -- Vamos a crear esta función para proteger el endpoint solicitando siempre que exista la sesión del usuario.
from todo.auth import login_required #Crearemos la función apenas terminemos de registrar el blueprint.
from todo.db import get_db

bp = Blueprint('todo', __name__) #Creamos el blueprint de nombre 'todo'.
```

```
@bp.route('/') #Este decorador se utiliza para indicar la URL que se debe asociar con la vista. En este caso, la URL es simplemente
@login_required #Una vez se ingrese en esta ruta, se chequea el inicio de sesión
def index(): #si el inicio de sesión es correcto, se activa la función index que listará los todo del usuario.
    db, c = get_db()
    c.execute(
        'select t.id, t.description, u.username, t.completed, t.created_at from todo t JOIN user u on t.created_by = u.id order by
        #Los elementos consultados (t.id, t.description, u.username, t.completed, t.created_at)
        #deben cumplir con que el created_by en la tabla de todo, sea igual al id en la tabla de user
        #y vendrán ordenados en base a created_at y de forma descendente.
    )
    todos = c.fetchall()
    return render_template('todo/index.html', todos = todos) #renderizar una plantilla HTML llamada "index.html" y enviarla como re
    #La plantilla "index.html" es renderizada con los valores pasados a través del diccionario todos = todos. El diccionario todos

@bp.route('/create', methods=['GET', 'POST']) #Creamos la ruta y asignamos los métodos.
@login_required #solicitamos que se cumpla login_required.
def create(): #definimos la función create.
    if request.method == 'POST': #verificamos que se esté usando metodo post
        description =  request.form['description'] # Tomamos la descripción desde request.form
        error = None #creamos un mensaje de error inofensivo.

        if not description: #Si la descripción no se ingresó
            error = 'Descripción es requerida' #Le indicamos al usuario el error.
        if error is not None: #Verificamos un cambio en error.
            flash(error) #Flasheamos un error
        else: #Si no hay error finalmente procedemos.
            db, c = get_db() # Obtenemos la bbdd y el cursor
            c.execute(
                #Ejecutamos un codigo SQL en la bbdd para insertar datos en la bbdd "todo".
                'insert into todo (description, completed, created_by)'
                'values (%s, %s, %s)',
                #Insertamos description como el input del usuario, false para indicar que la completitud sea 0, y g.user['id'] para
                (description, False, g.user['id'])
            )
            db.commit() #Comprometemos la bbdd.
            return redirect(url_for('todo.index')) #Redireccionamos al usuario a su pagina principal donde se muestran los ToDo (in
    return render_template('todo/create.html') #Asignamos la renderización de una plantilla en create.html

#Para esta edición vamos a indicar cual va a ser el ToDo que queremos actualizar.
# para esto primero indicamos que el id es un dato de tipo int.
def get_todo(id): #Definimos esta función para obtener el id del ToDo que vamos a editar.
#Recordemos que el id se lo estamos entregando a la función a través de la plantilla de index a la hora de seleccionar editar el To
    db, c = get_db() #Traemos nuestra base de datos y cursor.
    c.execute(
        #solicitamos los valores de id, descripción, completitud, autor y fecha de creación desde la tabla todo
        #Siempre y cuando el created_by corresponda al user id de quien inició la sesión.
        'select t.id, t.description, t.completed, t.created_by, t.created_at, u.username from todo t join user u on t.created_by =
        (id,)
    )
    todo = c.fetchone() #lanza el primer dato que encuentra.

    if todo is None:
        abort(404, "El ToDo de id{0} no existe".format(id)) #Caso en que no exista un todo para el id.

    return todo

@bp.route('/<int:id>/update', methods=['GET', 'POST'])
@login_required
def update(id):
    todo = get_todo(id) #Definimos el todo que vamos a editar

    if request.method == 'POST': #Verificamos que se este usando el metodo post
        description = request.form['description'] #guardamos la descripción
        completed = True if request.form.get('completed') == 'on' else False #Guardamos un valor booleano para el checkbox de compl
        error = None #creamos nuestra variable de error

        if not description: #Si description se encuentra vacío
            error = "La descripción es requerida."

        if error is not None: #Si descripción contiene un error
            flash(error)

        else: #Si no tenemos errores:
            db, c = get_db() #vamos a buscar a la bbdd y al cursor
            c.execute( #ejecutamos la actualización del ToDo
                'update todo set description = %s, completed = %s where id = %s',
                #Le entregamos un nuevo valor a description, completed para cuando se cumpla que el id sea igual al que tenemos alm
                (description, completed, id)
            )
            db.commit()
```

```
            return redirect(url_for('todo.index'))

    return render_template('todo/update.html', todo = todo)
```

4. Creamos la función para eliminar ToDo:

```
@bp.route('/<int:id>/delete', methods=['POST'])
@login_required
def delete(id):
    db, c = get_db()
    c.execute(
        'delete from todo where id = %s', (id,)
    )
    db.commit()
    return redirect(url_for('todo.index'))
```

5. Creamos el archivo html y editamos.

```
cd templates/todo
touch update.html
```

```
{% extends 'base.html' %}<!--Extendemos la plantilla desde base.html-->
{% block header %}
    <h1>{% block title %}Editar {{todo['description']}}{% endblock %}</h1> <!--Vamos a indicar el ToDo que queremos editar-->
{% endblock %}

{% block content %}
    <form method="post">
        <label for="description">Descripción</label><!--Solicitamos la nueva descripción del ToDo-->
        <!--Entregamos la caja para el nuevo input-->
        <!--En resumen, el atributo value se utiliza para establecer el valor predeterminado de un campo de entrada, y en este caso
        <input name="description" id="description" value="{{ request.form['description'] or todo['description']}}" required />
        <label for="completed">Completado</label><!--Revisaremos el valor de completitud del ToDo-->
        <input type="checkbox" name="completed" id="completed" {% if todo['completed'] == 1 %}checked{% endif %} />
        <input type="submit" value="Guardar" /><!--Entregamos un boton para guardar-->
    </form>
    <form action="{{url_for('todo.delete', id=todo['id']) }}" method="post"><!--Agregamos un boton para eliminar nuestro ToDo-->
        <input class="danger" type="submit" value="Eliminar" onclick="return confirm('¿Estás seguro de querer eliminarlo?')" />
    </form>
{% endblock%}
```

▼ Creamos una función para cerrar sesión.

En nuestro ‘base.html’ definimos que, si existe g.user, se le dará al usuario un link para cerrar sesión. Hasta aquí no hemos creado aún dicha ruta.

1. Vamos a ‘auth.py’:

```
import functools #set de funciones para construir aplicaciones.
from flask import(
    Blueprint, #Nos permite crear blueprints configurables.
    flash, #Funcion que nos permite enviar mensaje de forma generica a nuestras plantillas.
    g, #variable trascendental a la apricación.
    render_template, #renderiza plantillas
    request, #vamos a recibir datos desde un formulario.
    url_for, #vamos a crear urls.
    session, #para poder mantener una referencia del usuario que se encuentra en el contexto actual interactuando con la app.
    redirect
)
from werkzeug.security import (
    check_password_hash, #verifica que la contraseña que se ingresa sea igual a otra.
    generate_password_hash #encripta la contraseña que se está enviando.
)

from todo.db import get_db #importamos esto para poder interactuar con la bbdd.

bp = Blueprint('auth', __name__, url_prefix='/auth') #Es importante registrar este blueprint en la aplicación (__init__.py).

### - Agregamos una ruta de registro dentro de nuestro blueprint
```

```
@bp.route('/register', methods=['GET', 'POST']) #ruta /register, metodos que manejará la ruta.
def register():
    if request.method == 'POST': #Primero validamos dentro del request que se está enviando al sv sea el de POST.
        username = request.form['username'] #Vamos a obtener username.
        password = request.form['password'] #Vamos a obtener password.
        db, c = get_db() #Vamos a buscar la base y el cursor.
        error = None #definimos una variable de error para mas adelante usarla como flash.
        c.execute( #Ejecutamos la siguiente consulta a nuestra bbdd
            'select id from user where username = %s', #Esta consulta selecciona el id si el username en user es distinto de None.
            (username,) #Importante agregar el segundo argumento.
        )
# -- Definimos los 3 errores posibles para el registro de un usuario con 2 campos.
    # 1. Que no se registre un username.
        if not username: #Si no tenemos un username de parte del usuario entregamos un mensaje de error.
            error = 'Username es requerido'
    # 2. Que no se registre una password.
        if not password: #Si no tenemos un password de parte del usuario entregamos un mensaje de error.
            error = 'Password es requerido'
    # 3. Que el username registrado ya exista.
        elif c.fetchone() is not None: #Si, por el contrario, tenemos un username, vamos a buscar si existe la coincidencia dentro
            error = 'Usuario {} ya se encuentra registrado.'.format(username) #Esta sintaix con {} es solo posible si se utiliza el
# -- Validamos que el error no exista para ejecutar el registro.
        if error is None:
            c.execute(
                'insert into user (username, password) values (%s, %s)',
                (username, generate_password_hash(password)) #aquí llamamos el username y la contraseña encriptada con generate_pas
            )
            db.commit() #commit para comprometer la bbdd.

            return redirect(url_for('auth.login')) #Una vez que se haya completado el registro enviamos al usuario a auth.login (de

        flash(error) #Si no tuvimos un caso de éxito enviamos mensaje de error.

    return render_template('auth/register.html') #retornamos en caso de que la petición no sea POST sino GET.

### - Agregamos una ruta de login dentro de nuestro blueprint
@bp.route('/login', methods=['GET', 'POST']) #ruta /register, metodos que manejará la ruta.
def login():
    if request.method == 'POST': #Primero validamos dentro del request que se está enviando al sv sea el de POST.
        username = request.form['username'] #Vamos a obtener username.
        password = request.form['password'] #Vamos a obtener password.
        db, c = get_db() #Vamos a buscar la base y el cursor.
        error = None #definimos una variable de error para mas adelante usarla como flash.
        c.execute( #Ejecutamos la siguiente consulta a nuestra bbdd
            'select * from user where username = %s', (username,) #buscamos si existe algun usuario con un username coincidente cor
        )
        user = c.fetchone() #definimos la variable user como el registro coincidente de la bbdd con username.
# -- Verificamos errores
    # 1. Si no existe coincidencia en la bbdd.
        if user is None:
            error = 'Usuario y/o contraseña inválida' #Mensaje ambiguo para evitar obtener datos por la fuerza.
    # 2. Si existe coincidencia de usuario pero no de contraseña.
        elif not check_password_hash(user['password'], password):
            error = 'Usuario y/o contraseña inválida' #Mismo mensaje ambiguo para evitar entregar información específica respecto a

# -- En caso de que no exista error
        if error is None:
            session.clear() #Vamos a limpiar la sesión.
            session['user_id'] = user['id'] #creamos variable dentro de la sesión que vamos a llamar user_id, y le pasaremos el id
            return redirect(url_for('todo.index')) #redirigimos al usuario a la página de inicio.

        flash(error) # Caso en el que el usuario tuvo errores.

    return render_template('auth/login.html') #retornamos en caso de que la petición no sea POST sino GET.

### - FUNCION DECORADORA PARA CHEQUEAR EL LOGIN.
    # 1. Primero creamos una función que asigne el usuario dentro de la variable trascendental 'g'.
@bp.before_app_request
def load_logged_in_user(): #definimos la función.
    user_id = session.get('user_id') #vamos a sacar el user_id dentro de nuestro objeto de sesión.

    if user_id is None: #comprobamos si se obtuvo algo desde el objeto de sesión.
        g.user = None #si no se obtuvo nada, dejamos g.user como None.
    else:
        db, c = get_db() #Si existe, vamos a ir a buscar con el user_id el usuario a la bbdd.
        c.execute(
            'select * from user where id = %s', (user_id,) #busca en base al id que guardamos en sesión el registro en la bbdd.
        )
        g.user = c.fetchone() # Guarda en g.user el primer elemento.

    # 2. Definimos la función que protegerá los ToDos requiriendo el login.
```

```
def login_required(view):
    @functools.wraps(view) #Envolvemos a la función utilizando la función de wraps.
    def wrapped_view(**kwargs): #
        if g.user is None: #Preguntamos si g.user es None. Si es afirmativo, el usuario no ha iniciado sesión.
            return redirect(url_for('auth.login')) #Redireccionamos al usuario a la pagina de login.
        return view(**kwargs) #Si el usuario inició sesión, vamos a devolver la vista y los argumentos.
    return wrapped_view #Devolvemos la función recién creada.

### - FUNCIÓN DE LOGOUT.
@bp.route('/logout') #Creamos la ruta de logout
def logout():
    session.clear() #limpiamos la sesión del usuario.
    return redirect(url_for('auth.login')) #redirigimos al usuario al login.
```

▼ Lidiamos con errores de seguridad

- Al crear la función para eliminar ToDos, el único requerimiento que le entregamos fue que el usuario solo haya iniciado sesión. En este caso, cualquier usuario que inicie sesión podría eliminar cualquier ToDo.

```
#LTFlask/todo/todo.py

@bp.route('/<int:id>/delete', methods=['POST'])
@login_required
def delete(id):
    db, c = get_db()
    c.execute(
        'delete from todo where id = %s', (id,)
    )
    db.commit()
    return redirect(url_for('todo.index'))
```

- Debemos hacer que el usuario esté limitado a gestionar solo los ToDos que le pertenecen.

1. Actualizamos la función de delete (cambios en rojo).

   ANTES

   ```
   #LTFlask/todo/todo.py

   @bp.route('/<int:id>/delete', methods=['POST'])
   @login_required
   def delete(id):
       db, c = get_db()
       c.execute(
           'delete from todo where id = %s', (id,)
       )
       db.commit()
       return redirect(url_for('todo.index'))
   ```

   DESPUÉS

   ```
   #LTFlask/todo/todo.py

   @bp.route('/<int:id>/delete', methods=['POST'])
   @login_required
   def delete(id):
       db, c = get_db()
       c.execute(
           'delete from todo where id = %s and created_by = %s', (id, g.u
       )
       db.commit()
       return redirect(url_for('todo.index'))
   ```

2. Actualizamos la función update.

   ANTES

   ```
   #LTFlask/todo/todo.py

   @bp.route('/<int:id>/update', methods=['GET', 'POST'])
   @login_required
   def update(id):
       todo = get_todo(id) #Definimos el todo que vamos a editar

       if request.method == 'POST': #Verificamos que se este usando el metodo post
           description = request.form['description'] #guardamos la descripción
           completed = True if request.form.get('completed') == 'on' else False #Guardamos un valor booleano para el checkbox de
           error = None #creamos nuestra variable de error

           if not description: #Si description se encuentra vacío
               error = "La descripción es requerida."

           if error is not None: #Si descripción contiene un error
               flash(error)
   ```

```
        else: #Si no tenemos errores:
            db, c = get_db() #vamos a buscar a la bbdd y al cursor
            c.execute( #ejecutamos la actualización del ToDo
                'update todo set description = %s, completed = %s where id = %s',
                #Le entregamos un nuevo valor a description, completed para cuando se cumpla que el id sea igual al que tenemo
                (description, completed, id)
            )
            db.commit()
            return redirect(url_for('todo.index'))

    return render_template('todo/update.html', todo = todo)
```

3. Actualizamos la función index

   ANTES

```
#LTFlask/todo/todo.py

@bp.route('/') #Este decorador se utiliza para indicar la URL que se debe asociar con la vista. En este caso, la URL es simple
@login_required #Una vez se ingrese en esta ruta, se chequea el inicio de sesión
def index(): #si el inicio de sesión es correcto, se activa la función index que listará los todo del usuario.
    db, c = get_db()
    c.execute(
        'select t.id, t.description, u.username, t.completed, t.created_at from todo t JOIN user u on t.created_by = u.id orde
        #Los elementos consultados (t.id, t.description, u.username, t.completed, t.created_at)
        #deben cumplir con que el created_by en la tabla de todo, sea igual al id en la tabla de user
        #y vendrán ordenados en base a created_at y de forma descendente.
    )
    todos = c.fetchall()
    return render_template('todo/index.html', todos = todos) #renderizar una plantilla HTML llamada "index.html" y enviarla co
    #La plantilla "index.html" es renderizada con los valores pasados a través del diccionario todos = todos. El diccionario t
```

4.