PyMySQL

Para poder conectar nuestras aplicaciones Flask a una base de datos, necesitaremos un paquete que nos ayude. Utilizaremos uno llamado PyMySQL. A nivel de proyecto, ejecuta el siguiente comando en tu línea de comando para iniciar un nuevo proyecto.

new\_project\_folder $ pipenv install PyMySQL flask

Para cada proyecto que involucre una base de datos, realizaremos los siguientes pasos.

1. necesitaremos una base de datos a la que conectarnos. Creemos una en el entorno de trabajo llamada primer\_flask con una tabla única (mira el ERD a la derecha).
2. además del archivo server.py que creamos en cada proyecto, ahora también necesitaremos el siguiente archivo.

# un cursor es el objeto que usamos para interactuar con la base de datos

import pymysql.cursors

# esta clase nos dará una instancia de una conexión a nuestra base de datos

class MySQLConnection:

    def \_\_init\_\_(self, db):

        # cambiar el usuario y la contraseña según sea necesario DATOS DE MI CUENTA BASICAMENTE DE LARAGON QUEDA VACIO

        connection = pymysql.connect(

            host='localhost',

            user='root',

            password='root',

            db=db,

            charset='utf8mb4',

            cursorclass=pymysql.cursors.DictCursor,

            autocommit=True

        )

        # establecer la conexión a la base de datos

        self.connection = connection

    # el método para consultar la base de datos

    def query\_db(self, query, data=None):

        with self.connection.cursor() as cursor:

            try:

                query = cursor.mogrify(query, data)

                print("Running Query:", query)

                cursor.execute(query, data)

                if query.lower().find("insert") >= 0:

                    # las consultas INSERT devolverán el NÚMERO DE ID de la fila insertada

                    self.connection.commit()

                    return cursor.lastrowid

                elif query.lower().find("select") >= 0:

                    # las consultas SELECT devolverán los datos de la base de datos como una LISTA DE DICCIONARIOS

                    result = cursor.fetchall()

                    return result

                else:

                    # las consultas UPDATE y DELETE no devolverán nada

                    self.connection.commit()

            except Exception as e:

                # si la consulta falla, el método devolverá FALSE

                print("Something went wrong", e)

                return False

            finally:

                # cerrar la conexión

                self.connection.close()

# connectToMySQL recibe la base de datos que estamos usando y la usa para crear una instancia de MySQLConnection

def connectToMySQL(db):

    return MySQLConnection(db)

RESUMEN DEL CODIGO DE ARRIBA

* Las consultas SELECT devolverán una lista de diccionarios
* Las consultas INSERT devolverán el id generado automáticamente de la fila insertada
* Las consultas UPDATE y DELETE no devolverán nada
* Si la consulta falla, mal, devolverá False

1. Crear una clase que se asocie a la tabla con la que voy a trabajar. En el ejemplo de coding dojo:

first\_flask\_mysql/friend.py

# importar la función que devolverá una instancia de una conexión

from mysqlconnection import connectToMySQL

# modelar la clase después de la tabla friend de nuestra base de datos

class Friend:

    def \_\_init\_\_(self, data):

        self.id = data['id']

        self.first\_name = data['first\_name']

        self.last\_name = data['last\_name']

        self.occupation = data['occupation']

        self.created\_at = data['created\_at']

        self.updated\_at = data['updated\_at']

    # ahora usamos métodos de clase para consultar nuestra base de datos

    @classmethod

    def get\_all(cls):

        query = "SELECT \* FROM friends;"

        # asegúrate de llamar a la función connectToMySQL con el esquema al que te diriges

        results = connectToMySQL('first\_flask').query\_db(query) EL PROFE DIJO ESENCIAL

        # crear una lista vacía para agregar nuestras instancias de friends

        friends = []

        # Iterar sobre los resultados de la base de datos y crear instancias de friends con cls

        for friend in results:

            friends.append(cls(friend))

        return friends

1. Ahora vamos a actualizar nuestro archivo server.py para importar la clase y llamar al método de clase para consultar nuestra base de datos.

***first\_flask\_mysql/server.py***

    from flask import Flask, render\_template

# importar la clase de friend.py

from friend import Friend

app = Flask(\_\_name\_\_)

@app.route("/")

def index():

    # llamar al método de clase get all para obtener todos los amigos

    friends = Friend.get\_all()

    print(friends)

    return render\_template("index.html")

if \_\_name\_\_ == "\_\_main\_\_":

    app.run(debug=True)

**Recuperar y visualizar datos**

Cuando el usuario visite la ruta raíz, queremos mostrar todos los amigos, por lo que nuestra lógica incluirá buscar a todos los amigos de la base de datos y luego representar esos datos en nuestra plantilla.

***first\_flask\_mysql/server.py***

    @app.route('/')

def index():

    friends = Friend.get\_all()

    print(friends)

    return render\_template("index.html", all\_friends = friends)

***first\_flask\_mysql/templates/index.html***

<h1>Todos mis amigos</h1>

{% for one\_friend in all\_friends %}

    <p>First Name: {{one\_friend.first\_name}}</p>

    <p>Last Name: {{one\_friend.last\_name}}</p>

    <p>Occupation: {{one\_friend.occupation}}</p>

    <hr>

{% endfor %}

Con nuestra sentencia SELECT \* FROM friends , estamos pidiendo todo lo que hay en la base de datos. Sin embargo, a menudo querremos proporcionar datos variables en la consulta que serían terribles de codificar en la consulta. Por ejemplo:

* SELECT \* FROM friends WHERE id=1  (donde el número de id real variará)
* UPDATE friends SET first\_name="Bryanna" WHERE id=9 (donde el nombre y la identificación reales variarán)

**Siempre que ejecutamos una consulta que incluye variables, debemos usar una sentencia preparada en lugar de la interpolación de cadenas.** En otras palabras, utiliza el siguiente patrón en lugar de cadenas f o la concatenación de cadenas.

Prácticamente, lo que esto significa es que necesitaremos una **variable de cadena para la consulta** y luego un **diccionario para los valores que se utilizarán en la cadena**. Cuando llamamos a la conexión de la base de datos para ejecutar la consulta, pasaremos tanto la consulta como el diccionario, así:

*Para evitar que el usuario acceda a los datos de mi db, debo crear la consulta y una variable de consulta, entonces pasa por una especie de filtro que protege mi base de datos.*

query = "UPDATE friends SET first\_name=%(fn)s WHERE id=%(id\_num)s;"

data= {

    "fn": #possibly a value from a form;

    "id\_num": #possibly a value from the url

}

mysql.query\_db(query, data)

1. **query = "UPDATE friends SET first\_name=%(fn)s WHERE id=%(id\_num)s;"**: Esta línea define una consulta SQL de actualización. La consulta actualiza la columna **first\_name** en la tabla **friends** donde el **id** coincide con un valor proporcionado.
2. **data = { "fn": #posiblemente un valor de un formulario, "id\_num": #posiblemente un valor de la URL }**: Aquí se define un diccionario llamado **data**. Este diccionario contiene dos pares clave-valor. La clave **"fn"** representa el valor que se utilizará para actualizar la columna **first\_name**, y la clave **"id\_num"** representa el valor que se utilizará para encontrar la fila específica que se actualizará.
3. **mysql.query\_db(query, data)**: Esta línea ejecuta la consulta SQL utilizando la función **query\_db** de la instancia de la conexión a la base de datos MySQL llamada **mysql**. La consulta se ejecuta utilizando los valores proporcionados en el diccionario **data**. En este caso, **%(fn)s** y **%(id\_num)s** son marcadores de posición que serán reemplazados por los valores correspondientes del diccionario **data**.

TODO LO QUE SEA CONSULTA SQL DEBE TENER % Y S FUERA DEL PARENTESIS SI SE TRATA DE UNA VARIABLE, LO QUE VA DENTRO DEL PARENTESIS ES EL NOMBRE DE LA VARIABLE.

Y AFUERA HAY UN DICCIONARIO DE DATOS: LOS VALORES QUE SE INTERPOLARÁN EN LA CADENA DE CONSULTA. CLAVES DEL DICCIONARIO DE DATOS - FN, ID\_NUM: LAS CLAVES DEL DICCIONARIO DE DATOS UTILIZADAS EN LA CADENA DE CONSULTA CON %-INTERPOLACIÓN

query = "SELECT \* FROM users WHERE email = %(email)s;"

data = { 'email' : request.form['email'] }

result = mysql.query\_db(query, data)

Aquí, estamos usando una **sentencia preparada** para crear nuestra consulta SQL con los datos proporcionados por nuestro usuario. Esto se hace dejando marcadores de posición en nuestra consulta que se completan con los valores de nuestro diccionario de datos.

# este código es solo para fines de demostración

# NO use este código en producción, TE dejará vulnerable a la inyección SQL

query = f"SELECT \* FROM users WHERE email = '{request.form['email']}';"

result = mysql.query\_db(query)

Al usar la interpolación de cadenas simple, esto convertiría nuestra consulta en:

SELECT \* FROM users WHERE email = 'joe@gmail.com' OR '1'='1'

Dado que '1' = '1' siempre se evaluará como verdadero, esta consulta ahora obtendrá todos los datos de la tabla de usuarios. Es posible que hayamos abierto una gran parte de nuestra base de datos a un usuario malintencionado. Cualquier usuario con conocimientos de SQL puede descubrir fácilmente cómo manipular nuestras consultas SQL. Pueden obtener acceso a datos confidenciales u obligarnos a ejecutar una consulta muy peligrosa.

**Otros ejemplos de inyecciones SQL**

Considera otro escenario en el que el usuario ingresó lo siguiente como su correo electrónico:

joe@gmail.com"; DROP TABLE users;

¿Qué hubiera pasado si la forma en que preparaste la consulta SQL fuera así?

query = f"SELECT \* FROM users WHERE email = '{request.form['email']}';"

result = mysql.query\_db(query)

La consulta que habría ejecutado habría sido

SELECT \* FROM users WHERE email = "joe@gmail.com"; DROP users;

¿Qué pasa si el usuario pasó lo siguiente como entrada de correo electrónico?

joe@gmail.com"; UPDATE users SET password = '\_\_\_\_' WHERE id = '\_\_\_'

Esto habría cambiado la contraseña de alguien. De manera similar, puedes ver cómo uno podría configurarse como administrador o recuperar información confidencial de la tabla de otros usuarios (por ejemplo, tarjeta de crédito, dirección, etc.). Las posibilidades son infinitas.

***Sentencias preparadas al rescate***

query = "SELECT \* FROM users WHERE email = %(email)s;"

# la variable de marcador de posición se llama email

# debe coincidir con el nombre de la clave en el diccionario de datos

data = {

    # esta clave de 'email' en los datos debe tener un nombre que coincida con el marcador en la consulta

    'email' : request.form['email']

}

result = mysql.query\_db(query, data)

Ahora que podemos recuperar amigos, agreguemos la funcionalidad que nos permite crear un amigo. Primero necesitamos crear un **@classmethod** para crear una fila en la base de datos.

***first\_flask\_mysql/friend.py***

# importar la función que devolverá una instancia de una conexión

from mysqlconnection import connectToMySQL

# modelar la clase después de la tabla friend de nuestra base de datos

class Friend:

    # ...otros métodos de clase

    # método de clase para guardar a nuestro amigo en la base de datos

    @classmethod

    def save(cls, data ):

        query = "INSERT INTO friends ( first\_name , last\_name , occupation , created\_at, updated\_at ) VALUES ( %(fname)s , %(lname)s , %(occ)s , NOW() , NOW() );"

        # data es un diccionario que se pasará al método de guardar desde server.py

        return connectToMySQL('first\_flask').query\_db( query, data )

A continuación, agreguemos un formulario a nuestra plantilla:

***first\_flask\_mysql/templates/index.html***

<h1>Agregar un Amigo</h1>

<form action="/create\_friend" method="POST">

    <label for="fname">First Name:</label>

    <input type="text" name="fname">

    <label for="lname">Last Name:</label>

    <input type="text" name="lname">

    <label for="occ" >Occupation:</label>

    <input type="text" name="occ">

    <input type="submit" value="Add Friend">

</form>

Ahora necesitamos un método para manejar el envío de nuestro formulario. Actualicemos nuestro archivo **server.py** y pasemos request.form al método save de la clase Friend.

***first\_flask\_mysql/server.py***

# fragmento de código relevante de server.py

from friend import Friend

@app.route('/create\_friend', methods=["POST"])

def create\_friend():

    # Primero hacemos un diccionario de datos a partir de nuestro request.form proveniente de nuestra plantilla

    # Las claves en los datos tienen que alinearse exactamente con las variables en nuestra cadena de consulta

    data = {

        "fname": request.form["fname"],

        "lname" : request.form["lname"],

        "occ" : request.form["occ"]

    }

    # Pasamos el diccionario de datos al método save de la clase Friend

    Friend.save(data)

    # No olvides redirigir después de guardar en la base de datos

    return redirect('/')

Nota: Hemos configurado el método query\_db para que cada consulta probada se imprima en la terminal. *Siempre que la consulta que elaboraste parece no funcionar o muestra un mensaje de error, investiga la consulta real que se está ejecutando en la terminal. Puedes intentar copiar y pegar la consulta en MySQL Workbench para ver si tienes la sintaxis correcta.*

Modelo-vista-controlador (MVC)

Los desarrolladores que diseñan entornos de trabajo tienen que tomar decisiones sobre la organización del código. Uno de los patrones más populares para organizar código es uno conocido como MVC: **Modelo-vista-controlador**. Este es un desglose básico de responsabilidades:

|  |  |
| --- | --- |
| **Modelo** | 1. Puede construir tablas de bases de datos 2. Maneja la lógica que se basa en datos 3. Se interrelaciona con la base de datos |
| **Vista** | 1. Página HTML que se sirve al cliente 2. Puede contener cierta lógica para ser manejada por un motor de plantillas |
| **Controlador** | 1. Recibe solicitudes entrantes 2. Lógica mínima 3. Llama modelos para agregar/procesar datos 4. Determina la respuesta adecuada |

***Paso 1: la carpeta de la aplicación***

El proceso

1. Crea un directorio llamado **flask\_app**
2. Crea el archivo **\_\_init\_\_.py** dentro de la carpeta **flask\_app**
3. Inserta este código:

# \_\_init\_\_.py

from flask import Flask

app = Flask(\_\_name\_\_)

app.secret\_key = "shhhhhh"

1. Elimina las líneas anteriores de **server.py**
2. En **server.py** agregamos esta línea:

from flask\_app import app

# ...server.py

copy

1. Mover la carpeta templates a **flask\_app**
2. Mover la carpeta estática a **flask\_app**

***Paso 2: Controladores***

Ahora que tenemos nuestro módulo flask\_app configurado con plantillas y carpetas estáticas adentro, ahora necesitamos mover a él todo el código de nuestro server.py y organizarlo con un propósito.

El proceso

1. Crea carpetas de configuración y controladores dentro de la carpeta flask\_app.
2. Mueve mysqlconnection.py a la carpeta configuración.
3. Crea un archivo .py con el nombre de lo que estemos controlando en su forma plural.
4. Mueve todas las funciones @app.route al archivo del controlador.
5. Inserta este código en burgers.py:

# burgers.py

from flask\_app import app

from flask import render\_template,redirect,request,session,flash

from burger import Burger

1. Elimina las líneas anteriores de server.py
2. En server.py agregamos esta línea:

from flask\_app.controllers import burgers

# ...server.py

***Paso 3: Modelos***

Ahora que tenemos toda nuestra lógica en el archivo del controlador, burgers.py, podemos comenzar a separar las tareas del controlador con las del modelo.

Conclusiones clave

1. Utilizaremos nuestras habilidades de programación orientada a objetos (POO) para crear clases que se correspondan con las tablas de nuestra base de datos.
2. Usamos estas clases para consultar y mapear los datos brutos provenientes de la base de datos en objetos.
3. Nuestro archivo de controlador solo manejará la renderización, el redireccionamiento y la llamada a la clase Model para manejar la base de datos.

El proceso

1. Crea la carpeta modelos dentro de la carpeta flask\_app.
2. Mueve el archivo burger.py a los modelos.
3. Cambia la sentencia import en burger.py:

from flask\_app.config.mysqlconnection import connectToMySQL

# burger.py

class Burger:

def \_\_init\_\_(self,data):

self.id = data['id']

self.name = data['name']

self.bun = data['bun']

self.meat = data['meat']

self.calories = data['calories']

self.created\_at = data['created\_at']

self.updated\_at = data['updated\_at']

1. Haz que el controlador llame a los métodos de la clase Model.
2. Actualiza la sentencia import de Burger en el controlador

REPASO SOBRE CLASS METHOD: <https://docs.python.org/es/3/library/functions.html?highlight=classmethod#classmethod>

**Dividir las tareas entre modelos y controladores**

|  |  |
| --- | --- |
| **OBTENER TODAS LAS HAMBURGUESAS** | |
| CONTROLADORES | MODELOS |
| # burgers.py...  from flask\_app.models.burger import Burger  # obtiene todas las hamburguesas y las devuelve en una lista de objetos de hamburguesa  @app.route('/burgers')  *def* burgers():  return render\_templates('results.html',burgers=Burger.get\_all() | # burger.py...  # obtiene todas las hamburguesas y las devuelve en una lista de objetos de hamburguesa  @classmethod  *def* get\_all(cls):  query = "SELECT \* FROM burgers;"  burgers\_from\_db = connectToMySQL('burgers').query\_db(query)  burgers = []  for b in burgers\_from\_db:  burgers.append(cls(b))  return burgers |

|  |  |
| --- | --- |
| **CREAR UNA HAMBURGUESA** | |
| CONTROLADORES | MODELOS |
| # burgers.py...  from flask\_app.models.burger import Burger  # obtiene todas las hamburguesas y las devuelve en una lista de objetos de hamburguesa  @app.route('/create/burger',methods=['POST'])  def create\_burger():  data = {  "name" : request.form['name'],  "bun" : request.form['bun'],  "meat" : request.form['meat'],  "calories" : request.form['calories']  }  Burger.save(data)  return redirect('/burgers') | # burger.py...  # obtiene todas las hamburguesas y las devuelve en una lista de objetos de hamburguesa  @classmethod  def save(cls,data):  query = "Insert INTO burgers (name,bun,meat,calories,created\_at,updated\_at) VALUES(%(name)s,%(bun)s,%(meat)s,%(calories)s,NOW(),NOW());"  burger\_id = connectToMySQL('burgers').query\_db(query,data)  return burger\_id |

RELACIONES AVANZADAS

**Como puedes ver, necesitamos una tabla intermedia para relacionar el aderezo a la hamburguesa.  Una hamburguesa puede tener muchos aderezos y un aderezo puede estar en muchas hamburguesas.**

Ahora necesitamos crear nuestra clase Topping para alinearla con nuestra tabla Topping en nuestra base de datos. Junto con los métodos de clase para asociar nuestras clases.

**Utilizaremos esta consulta para obtener los datos que necesitamos para asociar nuestras Burgers a la clase Topping.**

"SELECT \* FROM toppings

LEFT JOIN add\_ons ON add\_ons.topping\_id = toppings.id

LEFT JOIN burgers ON add\_ons.burger\_id = burgers.id

WHERE toppings.id = %(id)s;"

***models/topping.py***

# necesitaremos importar burger.py para acceder a la clase

from flask\_app.models import burger

class Topping:

*def* \_\_init\_\_( self , db\_data ):

        self.id = db\_data['id']

self.topping\_name = db\_data['topping\_name']

        # necesitamos tener una lista en caso de que queramos mostrar qué hamburguesas están relacionadas con el aderezo

        self.on\_burgers = []

    self.created\_at = db\_data['created\_at']

self.updated\_at = db\_data['updated\_at']

    @classmethod

*def* save( cls , data ):

        query = "INSERT INTO toppings ( topping\_name, created\_at , updated\_at ) VALUES (%(topping\_name)s,NOW(),NOW());"

return connectToMySQL('burgers').query\_db(query, data)

    # este método recuperará el aderezo específico junto con todas las hamburguesas asociadas a él

    @classmethod

*def* get\_topping\_with\_burgers( cls , data ):

query = "SELECT \* FROM toppings LEFT JOIN add\_ons ON add\_ons.topping\_id = toppings.id LEFT JOIN burgers ON add\_ons.burger\_id = burgers.id WHERE toppings.id = %(id)s;"

results = connectToMySQL('burgers').query\_db( query , data )

# los resultados serán una lista de objetos topping (aderezo) con la hamburguesa adjunta a cada fila

topping = cls( results[0] )

for row\_from\_db in results:

# ahora parseamos los datos topping para crear instancias de aderezos y agregarlas a nuestra lista

burger\_data = {

"id" : row\_from\_db["burgers.id"],

"name" : row\_from\_db["name"],

               "bun" : row\_from\_db["bun"],

               "calories" : row\_from\_db["calories"],

"created\_at" : row\_from\_db["toppings.created\_at"],

"updated\_at" : row\_from\_db["toppings.updated\_at"]

}

topping.on\_burgers.append( burger.Burger( burger\_data ) )

return topping

Ahora agregamos un método de clase en nuestra clase burguer para asociar las clases.

**Utilizaremos esta consulta para obtener los datos que necesitamos para asociar nuestros aderezos a la clase Burger.**

"SELECT \* FROM burgers

LEFT JOIN add\_ons ON add\_ons.burger\_id = burgers.id

LEFT JOIN toppings ON add\_ons.topping\_id = toppings.id

WHERE burgers.id = %(id)s;"

***models/burger.py***

# necesitamos importar la clase topping de nuestros modelos

from flask\_app.models import topping

class Burger:

*def* \_\_init\_\_( self , data ):

        self.id = data['id']

self.name = data['name']

        self.bun = data['bun']

        self.meat = data['meat']

        self.calories = data['calories']

    self.created\_at = data['created\_at']

self.updated\_at = data['updated']

        # ahora creamos una lista para que luego podamos agregar todas los aderezos que están asociadas a una hamburguesa

        self.toppings = []

    # este método recuperará la hamburguesa con todos los aderezos asociados a la hamburguesa

@classmethod

*def* get\_burger\_with\_toppings( cls , data ):

query = "SELECT \* FROM burgers LEFT JOIN add\_ons ON add\_ons.burger\_id = burgers.id LEFT JOIN toppings ON add\_ons.topping\_id = toppings.id WHERE burgers.id = %(id)s;"

results = connectToMySQL('burgers').query\_db( query , data )

# los resultados serán una lista de objetos topping (aderezo) con la hamburguesa adjunta a cada fila

burger = cls( results[0] )

for row\_from\_db in results:

            # ahora parseamos los datos topping para crear instancias de aderezos y agregarlas a nuestra lista

topping\_data = {

"id" : row\_from\_db["toppings.id"],

"topping\_name" : row\_from\_db["topping\_name"],

"created\_at" : row\_from\_db["toppings.created\_at"],

"updated\_at" : row\_from\_db["toppings.updated\_at"]

}

burger.toppings.append( topping.Topping( topping\_data ) )

return burger

IMPORTACIONES CIRCULARES

Se llaman las instancias de forma doble y eso se debe evitar llamando al archivo, como sigue:  
Código refactorizado en topping.py

from flask\_app.models import burger  
topping.on\_burgers.append( burger.Burger( burger\_data ) )

Código refactorizado en burger.py

from flask\_app.models import topping  
burger.toppings.append( topping.Topping( topping\_data ) )

VALIDACION BASICA DE UN FORMULARIO

Las reglas de validación comunes incluyen:

* + Comprobar que los datos están presentes
  + Asegurarse de que los datos estén en el formato correcto

La herramienta de validación más importante es la sentencia if/else. ¡Cada validación es condicional! SI (if) los datos no son válidos, debemos avisar al usuario, DE LO CONTRARIO (else) debemos procesarlos y actualizar nuestra base de datos. La validación de formularios se centra en el uso de sentencias if combinadas con funciones que devuelven TRUE o FALSE dependiendo de si los datos proporcionados son válidos.

Paso 1: Método de validación en el modelo

Pongamos algunas validaciones en nuestra hamburguesa. Vamos a utilizar @staticmethod para validar los datos de nuestro formulario:

***models/burger.py***

from flask import flash

class Burger:

    # otros métodos Burger más allá

    # los métodos estáticos no tienen self o cls pasados a los parámetros

    # necesitamos tomar un parámetro para representar nuestra hamburguesa

    @staticmethod

*def* validate\_burger(burger):

        is\_valid = True # asumimos que esto es true

        if len(burger['name']) < 3:

            flash("Name must be at least 3 characters.")

            is\_valid = False

        if len(burger['bun']) < 3:

            flash("Bun must be at least 3 characters.")

            is\_valid = False

        if int(burger['calories']) < 200:

flash("Calories must be 200 or greater.")

is\_valid = False

        if len(burger['meat']) < 3:

 flash("Bun must be at least 3 characters.")

is\_valid = False

        return is\_valid

FLASH

Flash usa sesión para que podamos "proyectar" nuestros mensajes de error en la página del formulario.  Afortunadamente, **los mensajes flash son cadenas que existen para un solo ciclo de redireccionamiento**.  La diferencia entre Flash y sesión es que los mensajes flash solo duran una redirección, mientras que la sesión permanece hasta que se sale de forma manual. **Esto hace que los mensajes flash sean perfectos para validaciones en las que solo necesitamos mostrar el error o el mensaje temporalmente.**

Para utilizar flash, debemos importarlo a nuestros modelos. Bajo el capó, los mensajes flash utilizan sesión, por lo que debemos asegurarnos de que sesión esté configurada en nuestro \_\_init\_\_.py.

¡Ahora usar flash es tan fácil como invocar la función flash y pasar un mensaje de cadena! Primero veamos cómo se vería esto en la sentencia if. A continuación, veremos cómo mostrar los mensajes en el lado del cliente.

Paso 2: Pasar datos de formulario

En nuestro archivo de controlador, aceptamos un método POST y enviamos validar el formulario usando el Modelo:

***controllers/burgers.py***

from flask\_app.models.burger import Burger

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

*def* create\_burger():

# si hay errores:

    # llamamos al método estático en el modelo Burger para validar

 if not Burger.validate\_burger(request.form):

        # redirigir a la ruta donde se renderiza el formulario de burger

        return redirect('/')

# de lo contrario, no hay errores:

    Burger.save(request.form)

    return redirect("/burgers")

Paso 3: Visualización de mensajes flash en la plantilla

¡El último paso es incluir esos mensajes flash en la plantilla! La [documentación](http://flask.pocoo.org/docs/1.0/patterns/flashing/) nos muestra cómo:

{% with messages = get\_flashed\_messages() %}     <!-- declara una variable llamada mensajes

    {% if messages %}                            <!-- comprobar si hay algún mensaje -->

    {% for message in messages %}            <!-- recorrer los mensajes -->

    <p>{{message}}</p>                   <!-- muestra cada mensaje en una etiqueta de párrafo -->

        {% endfor %}

{% endif %}

{% endwith %}

PATRONES DE VALIDACION

Otra validación común que debe realizarse es verificar si una entrada coincide con un patrón determinado. Por ejemplo, las direcciones de correo electrónico tienen un patrón particular; a menudo se requiere que las contraseñas tengan un cierto número de tipos diferentes de caracteres. Podemos lograr esto usando algo conocido como **expresión regular** o **regex**.

Una expresión regular es una secuencia de caracteres que define un patrón de búsqueda. Se puede usar para hacer coincidir una cadena que sigue un patrón. Por ejemplo, cada correo electrónico tiene una serie de caracteres alfanuméricos seguidos de un símbolo @, seguido de otra serie de caracteres alfanuméricos, seguidos de un "." y, finalmente, otra serie de caracteres alfanuméricos. No necesitas saber cómo crear expresiones regulares en este punto, pero comprender qué son y para qué se utilizan es definitivamente importante. La expresión regular de Python para hacer coincidir una dirección de correo electrónico basada en los criterios anteriores se parece a esto (la r anterior indica que la cadena es una cadena sin procesar, es decir, todos los caracteres deben tomarse literalmente):

r'^[a-zA-Z0-9.+\_-]+@[a-zA-Z0-9.\_-]+\.[a-zA-Z]+$'

Los fragmentos a continuación muestran las adiciones importantes a nuestros modelos/archivo de usuario:

***models/user.py***

import re # el módulo regex

# crea un objeto de expresión regular que usaremos más adelante

EMAIL\_REGEX = re.compile(r'^[a-zA-Z0-9.+\_-]+@[a-zA-Z0-9.\_-]+\.[a-zA-Z]+$')

class User:

    @staticmethod

*def* validate\_user( user ):

        is\_valid = True

        # prueba si un campo coincide con el patrón

    if not EMAIL\_REGEX.match(user['email']):

    flash("Invalid email address!")

            is\_valid = False

        return is\_valid

El objeto EMAIL\_REGEX tiene un método llamado .match() que devolverá *None* si no se puede encontrar ninguna coincidencia. Si el argumento coincide con la expresión regular, se devuelve una instancia de objeto de coincidencia.

***controllers/users.py***

from flask\_app.models.user import User

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

*def* register():

    if not User.validate\_user(request.form):

        # redirigimos a la plantilla con el formulario

        return redirect('/')

    # ...hacer otras cosas

    return redirect('/dashboard')

Mensajes flash con categorías

Supongamos que queremos *clasificar* los mensajes flash con diferentes etiquetas o buckets (cubos). Podemos utilizar categorías pasando un segundo argumento en la función flash:

flash("Email cannot be blank!", 'email')

copy

Aquí está la [documentación sobre mensajes flash con categorías](http://flask.pocoo.org/docs/1.0/patterns/flashing/#flashing-with-categories), donde puedes obtener más información sobre ellos, incluido cómo mostrarlos en plantillas.

Otras herramientas de validación útiles:

* [str.isalpha()](https://docs.python.org/3.6/library/stdtypes.html#str.isalpha): devuelve un valor booleano que muestra si una cadena contiene solo caracteres alfabéticos
* [otros métodos de cadena](https://docs.python.org/3.6/library/stdtypes.html#string-methods)
* [time.strptime(string,format)](https://docs.python.org/3.6/library/time.html#time.strptime): cambia una cadena a una hora usando el formato dado

PROTECCION DE CONTRASEÑAS

Para esconder las contraseñas de nuestros usuarios, usaremos una técnica llamada *hashing*. El hash codifica los datos de tal manera que **no se puede revertir**. Esto es diferente de otra técnica de la que quizás hayas oído hablar llamada *encriptación*, que codifica datos con el uso de una clave de encriptación para que sea reversible. Con la encriptación, cualquiera que tenga la clave de cifrado puede descifrar los datos. Idealmente, solo el remitente y el receptor de los datos deberían conocer la clave de cifrado. Los datos de la tarjeta de crédito, por ejemplo, deben estar encriptados. Sin embargo, las contraseñas deben tener hash, y solo almacenaremos las contraseñas con hash en nuestra base de datos.

Una sal es una cadena larga aleatoria que agregamos a la contraseña de un usuario antes de aplicar el hash y almacenarla en la base de datos. Mantendremos la sal almacenada en nuestro servidor. Cuando el usuario regrese e inicie sesión, concatenaremos la sal a la entrada de la contraseña, la cifraremos con hash y veremos si coincide con el valor en la base de datos.

Solución: dar a cada usuario una sal única a través de Bcrypt

Con tu entorno pipenv activado (pipenv shell), ejecuta el siguiente comando:

pipenv install flask-bcrypt

En tu archivo modelo, agrega lo siguiente:

from flask\_bcrypt import Bcrypt

bcrypt = Bcrypt(app) # estamos creando un objeto llamado bcrypt,

# que se realiza invocando la función Bcrypt con nuestra aplicación como argumento

Luego de crear el objeto bcrypt, tenemos acceso a dos métodos que usaremos para generar nuestros hashes de contraseña y comparar contraseñas.

Para generar un hash, proporciona la contraseña que se usará como argumento.

bcrypt.generate\_password\_hash(“password\_string”)

Para comparar contraseñas, proporciona el hash como primer argumento y la contraseña que se comprobará como segundo argumento.

bcrypt.check\_password\_hash(hashed\_password, password\_string)

Exploremos el método generate\_password\_hash().Si le pasamos una cadena e imprimimos el resultado, es posible que veamos algo como esto:

$2b$12$sqjyok5RQccl9S6eFLhEPuaRaJCcH3Esl2RWLm/cimMIEnhnLb7iC

A continuación, cuando queremos verificar la contraseña de un usuario, vamos a compararla con el hash que hemos asociado a ese usuario en la base de datos. Pasamos tanto el hash y la contraseña proporcionada a check\_password\_hash(). Bcrypt extrae la sal a partir del hash y la aplica a la contraseña proporcionada, la cifra con hash, y compara el resultado con el hash guardado. Si coinciden, devuelve True. Si no, devuelve False.

Hash al registrarse

Veamos cómo se podría ver parte de nuestro código cuando estamos creando un usuario. En el siguiente ejemplo, no incluimos validaciones, pero, por supuesto que deberías validar la entrada del usuario antes de agregarla a la base de datos.

***controllers/users.py***

from flask\_app import app

from flask\_bcrypt import Bcrypt

bcrypt = Bcrypt(app)

@app.route('/register/user', methods=['POST'])

*def* register():

# validar el formulario aquí...

    # crear el hash

    pw\_hash = bcrypt.generate\_password\_hash(request.form['password'])

    print(pw\_hash)

    # poner pw\_hash en el diccionario de datos

    data = {

        "username": request.form['username'],

        "password" : pw\_hash

    }

# llama al @classmethod de guardado en Usuario

user\_id = User.save(data)

    # almacenar id de usuario en la sesión

    session['user\_id'] = user\_id

return redirect("/dashboard")

***models/user.py***

class User:

@classmethod

def save(cls,data):

query = "INSERT INTO users (username, password) VALUES (%(username)s, %(password)s);"

return connectToMySQL("mydb").mysql.query\_db(query, data)

Comparar al iniciar sesión

Así es como podría verse nuestro código cuando un usuario inicia sesión. Necesitaremos verificar la contraseña proporcionada con el hash en la base de datos:

***controllers/users.py***

from flask\_app import app

from flask\_bcrypt import Bcrypt

bcrypt = Bcrypt(app)

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

*def* login():

    # ver si el nombre de usuario proporcionado existe en la base de datos

data = { "email" : request.form["email"] }

    user\_in\_db = User.get\_by\_email(data)

    # usuario no está registrado en la base de datos

if not user\_in\_db:

        flash("Invalid Email/Password")

        return redirect("/")

    if not bcrypt.check\_password\_hash(user\_in\_db.password, request.form['password']):

    # si obtenemos False después de verificar la contraseña

        flash("Invalid Email/Password")

    return redirect('/')

    # si las contraseñas coinciden, configuramos el user\_id en sesión

    session['user\_id'] = user\_in\_db.id

    # ¡¡¡Nunca renderices en una post!!!

return redirect("/dashboard")

***models/user.py***

class User:

@classmethod

def get\_by\_email(cls,data):

query = "SELECT \* FROM users WHERE email = %(email)s;"

result = connectToMySQL("mydb").query\_db(query,data)

# no se encontró un usuario coincidente

if len(result) < 1:

return False

return cls(result[0])

.ENV

Archivo en donde puedo configurar todo lo que tenga que ver con la ubicación de la base de datos que estoy utilizando en mi proyecto, debe estar fuera de flask\_app, en el mismo nivel que server y eso y dentro pongo simplemente:

DATA\_BASE = aqui va la base de datos que estare usando

DATA\_BASE= BaseDeDatosUsada

USER\_DATA\_BASE=root

PASSWORD\_DATA\_BASE= 1234

SECRET\_KEY\_FLASK= contrasenhaSecreta