# RESUMEN:

En esta clase primero aprendimos sobre la librería **Flask-WTF**, que tiene una serie de ventajas a la hora de implementar un formulario en comparación a usar simple HTML.

* **Validación de formulario sencilla**: garantiza que el usuario ingrese datos en el formato requerido en todos los campos requeridos. Por ejemplo, comprobar que la entrada de correo electrónico del usuario tenga una "@" y un "." al final. Todo sin tener que escribir tu propio código de validación.
* **Menos código:** si tienes varios formularios en su sitio web, usar WTForm puede reducir drásticamente la cantidad de código que tiene que escribir (o copiar y pegar).
* **Protección CSRF incorporada:** CSRF significa Cross Site Request Forgery, es un ataque que se puede realizar en los formularios del sitio web y que obliga a los usuarios a realizar acciones no deseadas (por ejemplo, transferir dinero a un extraño) o comprometer la seguridad de su sitio web si es un administrador.

Luego vimos como Flask-WTF se integra con WTForms para crear formularios. Aquí un código de ejemplo de cómo crear una clase de un formulario con Flask-WTF y los campos del formulario con WTForms. Notar que la clase que creamos hereda de FlaskForm:

from flask\_wtf import FlaskForm

from wtforms import StringField, EmailField, PasswordField, SubmitField

class LoginForm(FlaskForm):

name = StringField(label='name')

email = EmailField(label='email')

pass = PasswordField(label='password')

submit = SubmitField(label='Log In')

Cada uno de los campos creados corresponde a un tipo específico disponible en la librería WTForms.

Una vez creada la clase podemos instanciarla y pasar el objeto a un template para renderizarlo. En nuestro ejemplo el objeto creado se llama login\_form:

@app.route("/login", methods=["GET", "POST"])  
def login():  
 login\_form = LoginForm()  
 return render\_template("login.html", form=login\_form)

En el template html recibimos el objeto form y llamamos a sus atributos para poder renderizarlos:

<form method="POST", action="{{ url\_for('login') }}" novalidate>  
 <p>  
 {{ form.email.label }} <br> {{ form.email }}  
 </p>  
 <p>  
 {{ form.password.label }} <br> {{ form.password }}  
 </p>  
 {{ form.submit }}  
</form>

Posteriormente revisamos como integrar validaciones y mostrar errores de validación. Para ello se realizaron varios cambios en el código.

* En el archivo principal se debe crear una llave privada y pasársela a app.secret\_key.
* Se debe importar wtforms.validators e implementar los validators en la definición de la clase del formulario.
* Luego de instanciar el formulario se debe llamar al método validate\_on\_submit() para que funcionen las validaciones luego de dar al botón submit del formulario. También la ruta que sirve este template debe poder procesar solicitudes de tipo POST.
* En el template html se debe crear el campo oculto {{ form.csrf\_token }} para implementar la protección csrf al formulario.
* Para visualizar nuestras validaciones correctamente se debe agregar el atributo novalidate al elemento form para deshabilitar las validaciones del navegador.
* En el mismo form se debe recorrer el listado de errores de validación mediante un ciclo for. Los errores luego se pueden mostrar al usuario en un elemento <span> por ejemplo.

from flask\_wtf import FlaskForm

from wtforms import StringField, EmailField, PasswordField, SubmitField

from wtforms.validators import DataRequired, Length, Email

class LoginForm(FlaskForm):

name = StringField(label='name', validators=[DataRequired()])

email = EmailField(label='email', validators=[DataRequired(),

Email()])

pass = PasswordField(label='password', validators=[DataRequired(), Length(min=8)])

submit = SubmitField(label='Log In')

app = Flask(\_\_name\_\_)  
SECRET\_KEY = os.urandom(32)

app.secret\_key = SECRET\_KEY

.

.

.

@app.route("/login", methods=["GET", "POST"])  
def login():  
 login\_form = LoginForm()

login\_form.validate\_on\_submit()  
 return render\_template("login.html", form=login\_form)

<form method="POST", action="{{ url\_for('login') }}" novalidate>  
 {{ form.csrf\_token }}  
 <p>  
 {{ form.email.label }} <br> {{ form.email(size=30) }}  
 {% for error in form.email.errors: %}  
 <span style="color: red">{{ error }}</span>  
 {% endfor %}  
 </p>  
 <p>  
 {{ form.password.label }} <br> {{ form.password }}  
 {% for error in form.password.errors: %}  
 <span style="color: red">{{ error }}</span>  
 {% endfor %}  
 </p>  
 {{ form.submit }}  
</form>

A continuación vimos que posterior a una solicitud de tipo POST podemos acceder a la información enviada en cada uno de los campos de un formulario simplemente llamando al objeto formulario y a la data en alguno de sus campos de esta forma <form\_object>.<form\_field>.data. En este ejemplo luego de validar que el método validate\_on\_submit() retorna verdadero accedemos a la data en los campos email y password del formulario y la comparamos a ciertos valores.

@app.route("/login", methods=["GET", "POST"])  
def login():  
 login\_form = LoginForm()  
 if login\_form.validate\_on\_submit():  
 if login\_form.email.data == "admin@email.com" and login\_form.password.data == *"12345678"*:  
 return render\_template("success.html")  
 else:  
 return render\_template("denied.html")  
 return render\_template("login.html", form=login\_form)

Luego vimos que es posible hacer herencia entre templates usando Jinja2. Para ello creamos una template base.html y definimos bloques de código vacío cuyo contenido puede ser personalizado en cada una de las templates que hereden de la plantilla base.

Para realizar la herencia primero se debe utilizar la expresión extends más el nombre de la plantilla base en cada una de las templates que van a heredar.

<!-- base.html -->

<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="utf-8">  
  
 <!-- empty block to be customized when a template inherits from this base template -->  
 <title>{% block title %}{% endblock %}</title>  
  
 <!-- Here we create a block that contains some code. If a template which inherits from this template wishes to use this code, then it can use the {{ super() }} expression -->  
 <style>  
 {% block styling %}  
 body{  
 background: black;  
 }  
 {% endblock %}  
 </style>  
  
</head>  
<body>

<!-- another empty block -->  
{% block content %} {% endblock %}  
  
</body>  
</html>

<!-- denied.html inherits from base.html -->

{% extends "base.html" %}  
{% block title %}Access Denied{% endblock %}  
  
<!-- uses {{ super() }} expression to keep the code definded for this block in the base.html template-->  
{% block styling %}  
 {{ super() }}  
 h1 {  
 color:red;  
 }  
{% endblock %}  
  
{% block content %}  
<div class="container">  
 <h1>Access Denied </h1>  
 <iframe src="https://giphy.com/embed/1xeVd1vr43nHO" width="480" height="271" frameBorder="0" class="giphy-embed"  
 allowFullScreen></iframe>  
 <p><a href="https://giphy.com/gifs/cheezburger-funny-dog-fails-1xeVd1vr43nHO">via GIPHY</a></p>  
</div>  
{% endblock %}

Finalmente vimos que existe la librería Bootstrap-Flask que permite integrar Bootstrap a un proyecto de Flask de forma fácil. Para ello se debe:

* Instalar la librería mediane pip install bootstrap-flask (NO CONFUNDIR CON la librería flask-bootstrap deprecada) e importar en el archivo main.py mediante from flask-bootstrap import Bootstrap5 (o 4 según se requiera).
* En la template base utilizar las funciones helper de Bootstrap-Flask para cargar tanto el CSS como el JS de Bootstrap5 (usar {{ bootstrap.load\_css() }} y {{ bootstrap.load\_js() }}, respectivamente).
* Luego de ello, en cualquier template que herede de la plantilla base tendremos a disposición distintos macros de Bootstrap-Flask como por ejemplo render\_form() que permiten con una sola línea renderizar un formulario completo y estilizado.
* Para poder utilizar estos macros primero debemos importarlos en la plantilla que hereda. Para el caso del macro render\_form() esto se haría mediante la expresión {% from "bootstrap5/form.html" import render\_form %}.
* Luego de importado el macro lo podemos utilizar pasándole el objeto form (login\_form en main.py) que hemos creado en el archivo main.py.

<!-- new base.html now using Bootstrap-Flask -->

<!doctype html>  
<html lang="en">  
<head>  
 {% block head %}  
 <!-- Required meta tags -->  
 <meta charset="utf-8">  
 <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">  
  
 {% block styles %}  
 <!-- loads Bootstrap CSS -->  
 {{ bootstrap.load\_css() }}  
 {% endblock %}  
  
 <title>{% block title %}{% endblock %}</title>  
 {% endblock %}  
</head>  
<body>  
{% block content %}{% endblock %}  
  
{% block scripts %}  
<!-- loads Bootstrap JavaScript -->  
{{ bootstrap.load\_js() }}  
{% endblock %}  
</body>  
</html>

<!-- login.html -->

{% extends "base.html" %}  
<!-- we import the render\_form() macro from Bootstrap-Flask -->  
{% from "bootstrap5/form.html" import render\_form %}  
{% block title %}Login{% endblock %}  
{% block content %}  
<div class="container">  
 <h1>Login</h1>  
 <!-- we pass the form to the render\_form() macro which renders the whole form element -->  
 {{ render\_form(form) }}  
{% endblock %}