

Facultad de Ciencias Exactas, Físicas y
Naturales

Departamento de Informática

Programación Orientada a Objetos

Unidad 5

- Licenciatura en Ciencias de la Computación
- Licenciatura en Sistemas de Información
- Tecnicatura Universitaria en Programación Web

Ciclo 2024

Contenido

1. Flask.....	3
2. La primera aplicación en Flask.....	3
2.1. Dando funcionalidad	3
2.2. Inicio del servidor web de Flask	4
3. El archivo de la aplicación Flask	4
4. Agregando funcionalidades.....	5
5. HTML en Flask	6
5.1. Rutas dinámicas.....	7
6. Formularios	7
Request.....	8
6.1. Envío de datos a un template.....	8
Jinja2.....	9
7. Plantilla base	11
8. Sesiones.....	12
9. Flask – CSS	14
10. Persistencia	17
11. Mapeo de objetos a base de datos relacional.....	17
12. SQLAlchemy.....	17
12.1. Uso de SQLAlchemy.....	18
12.2. Una aplicación	18
12.2.1. Clases involucradas en el incremento	18
Bibliografía.....	26
Enlaces de interés.....	26
Anexos.....	27
Conceptos de redes.....	27
Conceptos de Base de Datos	28

1. Flask

Flask es un framework, módulo de Python que permite desarrollar fácilmente aplicaciones web (Web Application Framework)¹, fue desarrollado por Armin Ronacher.

Se basa en el kit de librerías como Werkzeug y Jinja2 (proyectos de Pocco). **Werkzeug** es un toolkit para aplicaciones WSGI (Web Server Gateway Interface), que es una interface entre aplicaciones Python y servidores web. **Jinja2** es un motor para el renderizado de plantillas (templates) web. Flask es compatible con Python 3.8 y versiones posteriores.

2. La primera aplicación en Flask

Para escribir una aplicación Flask se puede usar cualquier editor de texto. Al ser un programa Python la extensión del archivo será .py.

Lo primero que se debe hacer es importar del framework la clase Flask, toda aplicación de Flask es una instancia de ella. El servidor web pasa todas las solicitudes que recibe de los clientes a este objeto para su manejo, utilizando un protocolo llamado *Web Server Gateway Interface* (WSGI). La instancia de la aplicación generalmente es creada de la siguiente manera:

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

El parámetro del constructor, `__name__` es el nombre del módulo Python actual. De este modo se le indica a Flask lo que pertenece a su aplicación. Este nombre se usa para buscar recursos en el sistema de archivos, las extensiones pueden usarlo para mejorar la información de depuración y mucho más.

2.1. Dando funcionalidad

Los navegadores web (clientes) envían solicitudes al servidor web, que a su vez las envía a la instancia de la aplicación Flask. La instancia de la aplicación necesita saber qué código debe ejecutarse para cada URL solicitada, por lo que mantiene una asociación entre la URL y la función de Python que la maneja, a esto se denomina **ruta**.

Para dar funcionalidad a una aplicación se debe asociar una ruta con una función. La forma de definir una ruta en una aplicación Flask es a través del decorador `@app.route`, que registra la función decorada como una ruta.

Para manejar el *path* o ruta raíz de la aplicación, el método `route` recibe como parámetro la '/', como se muestra a continuación:

¹ Web Application Framework o simplemente Web Framework representa una colección de bibliotecas y módulos que permiten a un desarrollador escribir aplicaciones web sin tener que preocuparse por detalles de bajo nivel como protocolos, gestión de subprocesos, etc.

```
@app.route('/')
def saludo():
    return 'Mi primera aplicación con Flask!'
```

De este modo la URL '/' en la aplicación registra la función **saludo()** como el manejador para la ruta raíz. Por lo tanto, cuando inicie la aplicación se ejecutará esta función.

Con la asociación de la ruta a una función, se manejan las peticiones (Request) y las respuestas (Response) para la aplicación. Las peticiones se resuelven en el servidor y las respuestas es lo que envía al cliente para que se muestre al usuario.

2.2. Inicio del servidor web de Flask

Una vez definida la estructura de la aplicación y las rutas que la conforman, otra sección importante de la aplicación es la que da inicio al servidor web de desarrollo integrado de Flask. La instancia de la aplicación posee el método `.run()` que inicia el servidor web de desarrollo integrado de Flask:

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

La implementación de la aplicación quedaría como se muestra a continuación.



```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def saludo():
    return 'Mi primera aplicación con Flask!'

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

El método `run`, posee argumentos que son opcionales, y permiten configurar el modo de operación del servidor web. Durante el desarrollo, es conveniente habilitar el modo de depuración, pasando el argumento de depuración establecido a `True` como se muestra a continuación:

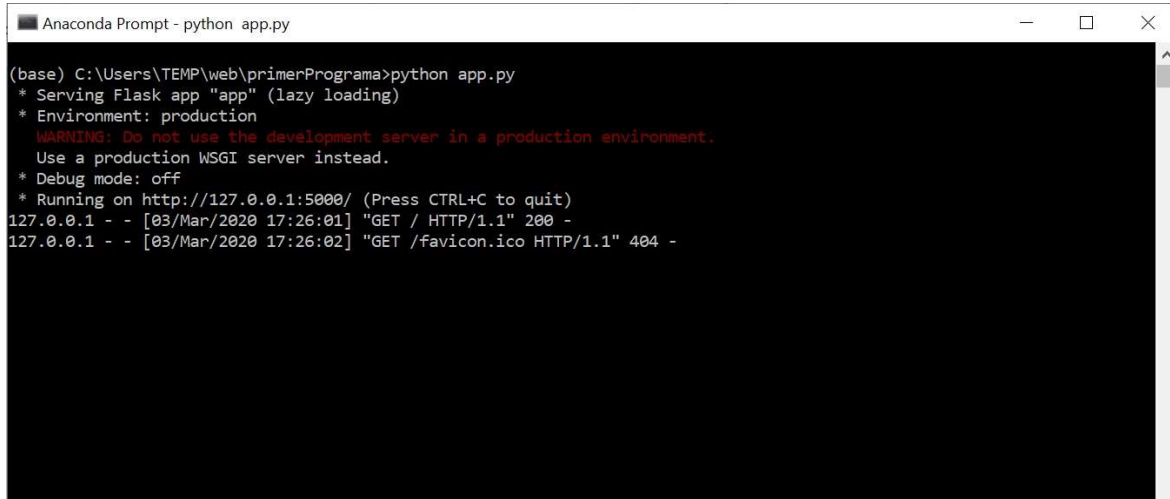
```
if __name__ == '__main__':
    app.run(debug=True)
```

3. El archivo de la aplicación Flask

Al ser una aplicación Python se debe guardar el archivo con una extensión `.py`. Por ejemplo: `app.py`. Luego se ejecuta el intérprete de Python y se solicita la ejecución de la aplicación.

```
$ python app.py
```

En la pantalla se visualizará que automáticamente inicia el servidor. Si la aplicación no posee errores, aparecerá una serie de mensajes como el que muestra la imagen.



```
Anaconda Prompt - python app.py
(base) C:\Users\TEMP\web\primerPrograma>python app.py
* Serving Flask app "app" (lazy loading)
* Environment: production
  WARNING: Do not use the development server in a production environment.
  Use a production WSGI server instead.
* Debug mode: off
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
127.0.0.1 - - [03/Mar/2020 17:26:01] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [03/Mar/2020 17:26:02] "GET /favicon.ico HTTP/1.1" 404 -
```

A partir de ese momento, si en la barra de direcciones del navegador se coloca la dirección `http://127.0.0.1:5000/` y la aplicación ejecutará.

4. Agregando funcionalidades

La aplicación web puede conformarse por varias funcionalidades, además de la inicial. Para agregar funcionalidades basta con crear otras rutas en la aplicación.

Por ejemplo, si queremos controlar la ruta `/saludo` el método `route` se invocará como:

```
@app.route('/saludo')
```

Un ejemplo de aplicación con más de una ruta se muestra en la siguiente imagen.



```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def saludo():
    return 'Mi primer aplicación Flask!'

@app.route('/bienvenida/')
def saludoBienvenida():
    return 'Bienvenido a mi aplicación Flask!'

@app.route('/despedida')
def saludoDespedida():
    return 'Gracias por visitar mi aplicación Flask!'

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

Al ejecutar la aplicación en el servidor web local, serían válidas las siguientes direcciones para el navegador:

- `http://127.0.0.1:5000`
- `http://127.0.0.1:5000/bienvenida`
- `http://127.0.0.1:5000/despedia`

Para el caso de “bienvenida”, también sería válida `http://127.0.0.1:5000/bienvenida/` por la forma como se ha declarado la ruta en el decorador `@app.route('/bienvenida/')`. En cambio, para “despedida”, la URL `http://127.0.0.1:5000/despedia/` produciría un error 404 porque la ruta se ha definido sin la “/” al final.

Si bien las aplicaciones vistas anteriormente, se han desplegado en un navegador web, no contienen páginas web.

5. HTML en Flask

HTML es la sigla de HyperText Markup Language, es un lenguaje de marcas estándar que permiten estructurar al contenido de la página web y es interpretado por los navegadores web.

A las marcas también se las conoce como etiquetas o tags. El estándar HTML especifica qué marcas deben utilizarse para escribir un documento web y qué significa cada una de ellas.

A continuación, se presenta un ejemplo de página web HTML.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Flask</title>
  </head>
  <body>
    <h1>Mi primera aplicación Flask</h1>
    <p>Esto es una página web</p>
  </body>
</html>
```

Para vincular una aplicación Flask con una página web se puede utilizar el método `render_template()`, que permite procesar un archivo HTML.

Si el HTML del ejemplo anterior se guarda en el archivo `inicio.html`, y se desea que sea la página web que se muestre cuando inicie la aplicación. Esto se debe especificar en la implementación de la aplicación donde está definida la ruta, como se muestra en el siguiente ejemplo:

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

Flask buscará el archivo HTML en la carpeta **templates**, la cual debe encontrarse dentro de la carpeta que contiene la aplicación `.py`.

5.1. Rutas dinámicas

Si se observan las URL de los servicios que se utilizan a diario se puede notar que muchas de ellas tienen secciones variables. Por ejemplo, la URL de una página de perfil de Facebook es `http://www.facebook.com/<your-name>`, donde el nombre de usuario forma parte de ella. Flask admite estos tipos de URL que contienen un componente dinámico utilizando una sintaxis especial en el decorador de ruta.

A diferencia de las rutas estáticas, las rutas creadas con reglas variables aceptan parámetros, y esos parámetros son las propias variables de ruta.

Entonces es posible construir una URL dinámicamente, agregando una parte variable al parámetro de la regla. Esta parte variable se indica con el nombre de la variable entre corchetes angulares `<nombre de variable>`, y se pasa como argumento a la función con la que está asociada la regla.

Si se desea que se muestre la página según el idioma, se puede pasar una cadena indicando el idioma, la cual podría ser una parte variable de la URL. Además, se convierte en parámetro de la función asociada a la regla. Es el caso del `<string:lenguaje>` en el ejemplo que se muestra a continuación.

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

@aplication.route('/')
@aplication.route('/<string:lenguaje>')
def saludo(lenguaje='es'):
    if lenguaje == 'es':
        return render_template('inicioes.html')
    else:
        return render_template('inicioen.html')
```

Las variables pueden incluir el tipo o no. Las rutas pueden aceptar los siguientes tipos de variables: string, int, float y path.

6. Formularios

Si el objetivo es crear una página web que permita ingresar datos que luego sean transferidos al servidor, allí se procesen y se retorne una respuesta al navegador, la página web deberá incluir una etiqueta HTML `<form>`. Como se presenta en la siguiente imagen.

```
<!DOCTYPE html>
<html>
  <body>
    <h3>Registrar Usuario</h3>
    <hr/>
    <form action = "http://localhost:5000/alta" method = "post">
      <label for = "name">Nombre</label><br>
      <input type = "text" name = "name" /><br>
      <label for = "email">Email</label><br>
      <input type = "text" name = "email" /><br>
      <label for = "clave">Contraseña</label><br>
      <input type = "text" name = "clave" /><br>
      <input type = "submit" value = "Submit" />
    </form>
  </body>
</html>
```

La comunicación entre el cliente y el servidor se realiza a través del protocolo HTTP, que es la base de la comunicación de datos en la red mundial. En este protocolo se definen diferentes métodos de recuperación de datos de la URL especificada.

La comunicación de los datos tiene dos posibles métodos GET y POST. Una página HTML que contiene un formulario, en la etiqueta <form> a través de la propiedad method especifica el método (GET o POST) que empleará para la comunicación.

En el ejemplo mostrado en la imagen anterior se usa el método post para el envío del formulario, tal como lo indica la línea:

```
<form action = "http://localhost:5000/alta" method = "post">
```

Request

Los datos de la página web cliente se envían al servidor encapsulados en un objeto request global. Para utilizar este objeto se debe importar.

```
from flask import request
```

El objeto request posee el atributo method que permite saber el tipo de petición que hace el cliente (GET o POST). Es posible hacer el siguiente análisis:

```
if request.method == 'POST':
```

6.1. Envío de datos a un template

Cuando se define una ruta, el método route(), recibe como parámetro el string con la ruta, pero además, puede recibir otros parámetros, como por ejemplo el método (GET y/o POST) con el que operará.

Los datos del formulario recibido por la función activada pueden ser recuperados en forma de un objeto y reenviarse a un *template* para representarlos en la página web correspondiente.

Jinja2

El motor de template **Jinja2** que incluye Flask, permite que la página web receptora de los datos los use a través del delimitador `{{...}}` para escapar de HTML. Este delimitador se usa para que las expresiones se impriman en la salida del template.

Otro delimitador de interés es `{% ... %}`, que se usa para incluir en el template expresiones que alteran el flujo secuencial del HTML, como son los condicionales e iteraciones.

Ejemplo

El jefe de proyecto nos ha solicitado implementar parte de una aplicación web. La solicitud consiste en una página web que permita ingresar el nombre, correo y contraseña de un usuario. Si se han ingresado los tres datos se debe dirigir a otra página que muestre un saludo que contenga el nombre y el correo del usuario. En caso de faltar algún dato se debe redirigir a la página de ingreso.

La resolución planteada presenta la siguiente lógica:

- La URL `/` muestra una página web (`nuevo_usuario.html`) que tiene un formulario. Los datos ingresados se publican en la URL `/bienvenida` que activa la función `bienvenida()`.
- La función `bienvenida()` recopila los datos del formulario a través del objeto `request` y los envía a `bienvenida.html` para mostrar el mensaje de bienvenida.

A continuación se muestra el código de aplicación en Python que guardamos en el archivo **app.py**

```
1 from flask import Flask, render_template, request
2 from datetime import datetime
3 app = Flask(__name__)
4
5 @app.route('/')
6 def usuario():
7     return render_template('nuevo_usuario.html')
8
9 @app.route('/bienvenida', methods = ['POST', 'GET'])
10 def bienvenida():
11     if request.method == 'POST':
12         if request.form['nombre'] and request.form['email'] and
13 request.form['password']:
14             datosf = request.form
15             return render_template('bienvenida.html', datos=datosf, hora =
16 datetime.now().hour)
17         else:
18             return render_template('nuevo_usuario.html')
19 if __name__ == '__main__':
20     app.run(debug = True)
```

nuevo_usuario.html

```
1 <!DOCTYPE html>
2 <html>
3     <body>
4         <h3>Nuevo Usuario</h3>
5         <form action = "http://localhost:5000/bienvenida" method = "post">
6             <label for = "nombre">Nombre</label><br>
```

```

7      <input type = "text" name = "nombre" placeholder = "Nombre" /><br>
8
9      <label for = "email">Correo Electrónico</label><br>
10     <input type = "text" name = "email" placeholder = "email" /><br>
11
12     <label for = "password">Contraseña</label><br>
13     <input type = "password" name = "password" placeholder = "password"/><br>
14     <br>
15     <input type = "submit" value = "Guardar" />
16     <br><br>
17 </form>
18 </body>
19 </html>

```

bienvenida.html

```

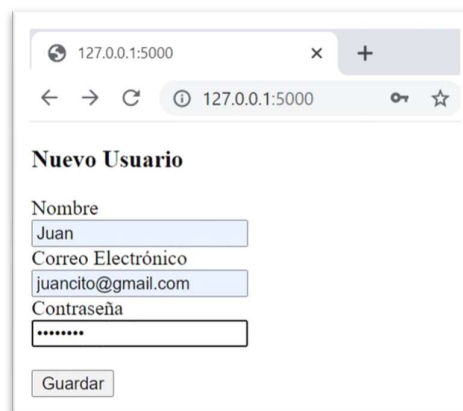
1 <!doctype html>
2 <html>
3   <body>
4       {% if hora < 12 %}
5           <h1> Buenos días {{ datos.nombre }} </h1>
6       {% elif hora < 21 %}
7           <h1> Buenas tardes {{ datos.nombre }} </h1>
8       {% else %}
9           <h1> Buenas noches {{ datos.nombre }} </h1>
10      {% endif %}
11      <h2> Recibirás las notificaciones en el correo {{ datos.email }}</h2>
12
13   </body>
14 </html>

```

Las líneas 4, 6, 8 y 10 conforman una estructura de selección a través del uso del delimitador {% %}.

En las líneas 5, 7, 8 y 11 a través del delimitador {{...}} se inserta en el HTML de salida los datos recibidos.

Ejecutamos la aplicación en Python e ingresamos la URL <http://localhost:5000/> en el navegador.



The screenshot shows a web browser window with the address bar displaying '127.0.0.1:5000'. The page content is a form titled 'Nuevo Usuario'. It contains three input fields: 'Nombre' with the value 'Juan', 'Correo Electrónico' with the value 'juancito@gmail.com', and 'Contraseña' with masked characters '.....'. Below these fields is a button labeled 'Guardar'.

Cuando se hace clic en el botón “Guardar”, se ejecuta la función bienvenida definida en la línea 10 del archivo app.py. Evalúa que se haya ingresado un valor en los campos (línea 12 de app.py), en caso de cumplirse esta condición envía los datos del formulario recibido (línea 15 de app.py) a la página bienvenida.html.



En caso de que la evaluación de los datos del formulario sea falsa se redirige a la página nuevo_usuario.html (línea 18 de app.py).

7. Plantilla base

Es muy común que una aplicación posea un conjunto de páginas que comparten la misma estructura. Para evitar repetir código es posible crear una plantilla base que pueda ser reutilizada.

Para crear plantillas y bloques de código reutilizables, Jinja2 provee la etiqueta `{% block %}{% endblock %}`

Esta plantilla es un archivo .html que incluirá las etiquetas de bloques, como muestra la siguiente imagen que corresponde al archivo base_template.html.

```
1 <!DOCTYPE html>
2 <html lang="es">
3 <head>
4   <meta charset="UTF-8">
5   <title>{% block title %}{% endblock %}</title>
6   <link rel="stylesheet" href="{{ url_for('static', filename='css/estilos.css') }}">
7 </head>
8 <body>
9   {% block content %}{% endblock %}
10   <div class="myDiv">
11     <br><br><a href = "{{ url_for('inicio') }}" > Inicio </a><br><br>
12   </div>
13 </body>
14 </html>
```

En la línea 5 hay un bloque para el título y en la línea 9 un bloque para el contenido de la página.

Un archivo html que reutilice una plantilla base deberá indicarlo a través de la etiqueta `{% extends %}` y además incluir el contenido para cada bloque definido en la plantilla base.

A continuación se muestra el archivo nuevo_usuario.html que hace uso de la plantilla base base_template.html.

```

1  {% extends "base_template.html" %}
2  {% block title %}Crear Usuario{% endblock %}
3  {% block content %}
4      <h1>Registro de Usuario</h1>
5      <hr/>
6      <form action = "{{ request.path }}" method = "post" class = "container">
7          <label for = "nombre">Nombre</label><br>
8          <input type = "text" name = "nombre" placeholder = "Nombre" /><br>
9
10         <label for = "email">Correo Electrónico</label><br>
11         <input type = "text" name = "email" placeholder = "email" /><br>
12
13         <label for = "password">Contraseña</label><br>
14         <input type = "password" name = "password" placeholder = "password"/><br>
15         <br>
16         <input type = "submit" value = "Guardar" />
17     </form>
18 {% endblock %}

```

8. Sesiones

La sesión es el intervalo de tiempo que transcurre desde que un cliente ingresa a la aplicación (inicia sesión) hasta que cierra la aplicación. Cada cliente tendrá su propia sesión donde sus propios datos serán almacenados en un directorio temporal en el servidor.

Se asigna un ID de sesión a cada sesión de un cliente. Cuando los datos de la sesión son almacenados, el servidor los firma en modo cifrado, para esto, la aplicación Flask requiere que se defina una SECRET_KEY.

Para manejar sesiones en Flask, la aplicación debe usar la clase session. Una sesión es un objeto diccionario que contiene pares de valores, *clave* para variables de sesión y *valores* asociados. Por ejemplo:

```
session['username'] = 'admin'
```

Para liberar una variable de sesión se usa el método pop(). Por ejemplo:

```
session.pop('username', None)
```

A continuación se presenta un ejemplo simple donde se guarda el nombre del usuario en un objeto session.

El archivo de la aplicación debe importar la clase session. Además, define SECRET_KEY (línea 4). Establece 3 rutas: la ruta principal ("/") en la línea 6; la ruta de login (línea 13) y la de logout en la línea 21.

En la línea 8 (dentro de la función que resuelve la ruta principal) se verifica si existe la variable de sesión "name".

En la línea 16 (dentro de la función que resuelve la ruta del login) se le asigna al objeto sesión, en la variable "name", el valor del elemento HTML llamado "name" que incluye el formulario.

En la línea 23 (dentro de la función que resuelve la ruta del logout) se le asigna al objeto sesión, en la variable "name", el valor None.

```

1  from flask import Flask, render_template, redirect, request, session
2
3  app = Flask(__name__)
4  app.secret_key = 'CLAVE_SECRETA'
5
6  @app.route("/")
7  def index():
8      if not session.get("name"):
9          return redirect("/login")
10         return render_template('index.html')
11
12
13  @app.route("/login", methods=["POST", "GET"])
14  def login():
15      if request.method == "POST":
16          session["name"] = request.form.get("name")
17          return redirect("/")
18          return render_template("login.html")
19
20
21  @app.route("/logout")
22  def logout():
23      session["name"] = None
24      return redirect("/")
25
26
27  if __name__ == "__main__":
28      app.run(debug=True)
29

```

En los templates, que se definen son:

La plantilla base con nombre de archivo plantilla.html.

```

1
2  <!DOCTYPE html>
3
4  <html lang="es">
5      <head>
6          <meta name="viewport" content="initial-scale=1, width=device-width">
7          <title> Sesión </title>
8      </head>
9      <body>
10         {% block y %}{% endblock %}
11     </body>
12 </html>

```

El template para la ruta principal es el archivo index.html

```

1  {% extends "plantilla.html" %}
2
3  {% block y %}
4
5      {% if session.name %}
6          Usted se ha registrado como {{ session.name }} <br> <br> <a href="/logout">Cerrar sesión</a>.
7      {% else %}
8          No se ha registrado. <a href="/login">Iniciar sesión</a>.
9      {% endif %}
10
11  {% endblock %}
12

```

En la línea 6 se escribe el valor de la variable de sesión.

El template login.html, que muestra el formulario donde el usuario ingresa su nombre y lo envía al servidor.

```

1  {% extends "plantilla.html" %}
2
3  {% block y %}
4
5  <h1> Iniciar Sesión </h1>
6
7  <form action="/login" method="POST">
8      <input placeholder="Nombre" autocomplete="off" type="text" name="name">
9      <br> <br>
10     <input type="submit" name="Iniciar">
11 </form>
12
13 {% endblock %}
14

```

9. Flask – CSS

Para agregar estilos a la estructura HTML usando CSS (hojas de estilo en cascada) es necesario crear un archivo CSS y conectarlo a los archivos HTML.

Las hojas de estilo CSS se consideran archivos estáticos. No hay interacción con el código, como ocurre con las plantillas HTML. Por lo tanto, Flask ha reservado una carpeta separada donde se debe colocar los archivos estáticos como CSS, JavaScript, imágenes u otros archivos. Esa carpeta debe llamarse **static**. También es una buena práctica crear dentro de static una carpeta y nombrarla css.

La vinculación de una hoja de estilos CSS con un HTML se efectúa en la sección <head> del HTML a través de una etiqueta <link>. Esta etiqueta requiere la dirección del archivo .css, para ello se usa la función url_for que genera la URL. Como se trata de un archivo estático el primer parámetro de la función será 'static' y el segundo el nombre del archivo.

Ejemplo

Al ejemplo visto anteriormente le incluiremos una hoja de estilos CSS.

Estilos.css

```

body {
    margin: 0;
    padding: 0;
    font-family: "Helvetica Neue", Helvetica, Arial,
    sans-serif;
    color: #464;
}

header {
    background-color: #DFB887;
    height: 10%;
    width: 100%;
    opacity: .9;
    margin-bottom: 2%;
}

h1 {
    font-family: serif;
    text-align: center;
    color: #377ba8;
}

.container {

```

```

padding-top: 2%;
width: 50%;
margin: 0 auto;
background-color: #DFB887;
}

.container label {
font-weight: bold;
margin-bottom: 2em;
margin-left: 2em;
color: #377ba8;
}

.container input {
margin-bottom: 1em;
margin-left: 2em;
}

.container button {
margin-bottom: 1em;
margin-left: 2em;
}

.container textarea {
min-height: 12em;
resize: vertical;
}

```

nuevo_usuario.html

```

1 <!DOCTYPE html>
2 <html>
3   <head>
4     <title>Crear Usuario</title>
5     <link      rel="stylesheet"      href="{{url_for('static',      filename=
6 'css/estilos.css')}}">
7   </head>
8   <body>
9     <h1>Nuevo Usuario</h1>
10    <form action = "http://localhost:5000/bienvenida" method = "post"
11    class= "container">
12      <label for = "nombre" >Nombre</label><br>
13      <input type = "text" name = "nombre" placeholder = "Nombre" /><br>
14
15      <label for = "email">Correo Electrónico</label><br>
16      <input type = "text" name = "email" placeholder = "email" /><br>
17
18      <label for = "password">Contraseña</label><br>
19      <input type = "password" name = "password" placeholder =
20 "password"/><br>
21      <br>
22      <input type = "submit" value = "Guardar" />
23      <br><br>
24    </form>
25  </body>
</html>

```

La aplicación ahora desplegaría la página como muestra la siguiente imagen.

Crear Usuario

127.0.0.1:5000

Nuevo Usuario

Nombre

Nombre

Correo Electrónico

email

Contraseña

password

Guardar

Flask – SQLAlchemy

10. Persistencia

Es muy común que una aplicación desarrollada bajo el paradigma de la orientación a objetos requiera la persistencia de los objetos que maneja. Es decir, es necesario almacenar los objetos en un medio (por ejemplo, el disco rígido) que permita recuperarlos en cualquier momento futuro, independientemente de la ejecución de la aplicación. Para esto, se suele usar base de datos relacionales, donde la información se guarda en tablas formadas por filas y columnas.

Es necesario aclarar que el diseño y administración de una base de datos no es objetivo de esta asignatura, por lo que, solo se verán los conceptos y mecanismos mínimos necesarios para lograr que una aplicación almacene y recupere información de una base de datos relacional (en sección de anexo se incluyen conceptos de base de datos).

La aplicación orientada a objetos debe realizar una serie de operaciones sobre el conjunto de tablas que contiene la base de datos, para ello deben comunicarse entre sí, convirtiéndose esto en un inconveniente que se puede resolver con librerías de mapeo objeto – relacional.

11. Mapeo de objetos a base de datos relacional

Una API ORM (Object Relational Mapping) proporciona métodos para traducir la lógica de los objetos a la lógica relacional sin tener que escribir instrucciones en el lenguaje de consulta de base de datos (SQL - Structured Query Language).

Las ventajas de usar un ORM son:

- Independencia de la aplicación con el motor de datos empleado en producción.
- Soporta los distintos motores de base de datos.
- Cambiando una línea en el archivo de configuración ya se puede emplear una base de datos distinta.

Distintos paquetes han sido desarrollados para facilitar el proceso de mapeo de objetos a base de datos relacional, proveyendo bibliotecas de clases que son capaces de realizar mapeos automáticamente.

12. SQLAlchemy

Es un ORM que provee mecanismos para asociar clases Python definidas por el usuario con tablas de bases de datos, e instancias de esas clases (objetos) con filas (registros) en sus tablas correspondientes. Incluye un sistema que sincroniza de manera transparente todos los cambios de estado entre los objetos y sus filas relacionadas, denominado unidad de trabajo. Además de un sistema para expresar consultas de la base de datos en términos de las clases definidas por el usuario y sus relaciones definidas entre sí.

SQLAlchemy proporciona una interfaz estándar que permite a los desarrolladores crear código independiente de la base de datos para comunicarse con una amplia variedad de motores de bases de datos. Se admiten los sistemas de administración de bases de datos más comunes disponibles. PostgreSQL, MySQL, Oracle, Microsoft SQL Server y SQLite son ejemplos de motores que se pueden usar con SQLAlchemy.

Para desarrollar aplicaciones que utilicen Flask-SQLAlchemy el primer paso es instalarlo, ya que no viene instalado por defecto (*pip install flask-sqlalchemy*).

12.1. Uso de SQLAlchemy

SQLAlchemy contiene funciones para operaciones ORM. También proporciona una clase `Model` que es la clase base para generar los modelos definidos por el usuario para el mapeo.

SQLAlchemy gestiona las operaciones de persistencia a través de distintos objetos. Esto es, un objeto SQLAlchemy, un objeto sesión y sus respectivos métodos. Algunos de los métodos más comunes para realizar operaciones sobre la base de datos son:

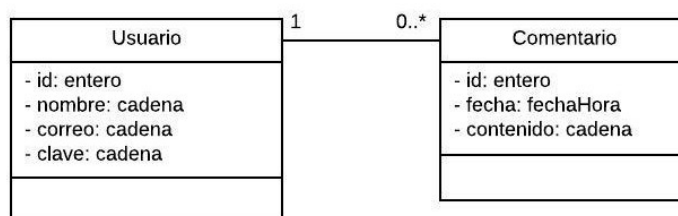
- ✓ `add`: inserta un registro en la tabla correspondiente de acuerdo al objeto que se pase como parámetro.
- ✓ `delete`: elimina el registro de la tabla correspondiente según el objeto que se pase como parámetro.
- ✓ `query.all()`: recupera todos los registros de la tabla correspondiente según el objeto `model` al que se aplique. Flask-SQLAlchemy provee un objeto `query` para cada clase del modelo. Además, a través de un objeto `query` se puede aplicar un filtro al conjunto de registros recuperados.

12.2. Una aplicación

La empresa de software donde trabajan está desarrollando una aplicación Flask para el dictado de un curso on-line con usuarios registrados. Les pide a ustedes que desarrollen el incremento de “Comentarios” para dicha aplicación, cumpliendo los siguientes requerimientos o requisitos:

- R1. Registrar un comentario que realiza un usuario registrado.
- R2. De un comentario se registra la fecha y hora en que se realiza, el texto del comentario y el usuario que lo realiza.
- R3. Un usuario puede hacer varios comentarios.
- R4. Al momento de realizar un comentario, el usuario debe ingresar su correo y contraseña, si estos datos se corresponden con un usuario registrado se le permite comentar.
- R5. Listar todos los comentarios incluyendo los datos de los usuarios que lo hizo.
- R6. Listar los comentarios para un usuario en particular.

12.2.1. Clases involucradas en el incremento



A continuación se describe el desarrollo de lo solicitado para la aplicación Flask-SQLAlchemy.

Paso 1

Establecer los parámetros de configuración referidos a la base de datos (la URI de la base de datos, clave en caso de que la tenga, etc.) que se utilizará, la sesión, etc. Si bien, es posible incluirlos en la implementación de la propia aplicación, es conveniente generar un archivo

independiente con estos valores. Para sqlite3 solo es necesario establecer la URI de la base de datos.

El contenido del archivo será el siguiente:

```
SECRET_KEY = "GDtfdCFYjD"  
SQLALCHEMY_DATABASE_URI = 'sqlite:///datos.sqlite3'  
SQLALCHEMY_TRACK_MODIFICATIONS = False
```

El parámetro SECRET_KEY, corresponde a una clave secreta que se usará en ciertas operaciones que requieren cierta seguridad, por ejemplo cuando se crea una sesión.

El parámetro SQLALCHEMY_DATABASE_URI indica el nombre del archivo que contiene la base de datos. En este es una base de datos que corresponde al administrador sqlite y se llama datos.sqlite3.

El parámetro SQLALCHEMY_TRACK_MODIFICATIONS si se establece en True, Flask-SQLAlchemy rastreará las modificaciones de los objetos y emitirá señales. Esto requiere memoria adicional y debe deshabilitarse si no es necesario.

Nota: Para este caso de aplicación el archivo se guarda con el nombre **config.py**.

Paso 2

Definir el modelo para la base de datos a través de una clase para cada tabla. En primer lugar es necesario recuperar desde la aplicación principal la instancia que la representa. Luego se crea una instancia de la clase SQLAlchemy, la cual será la proveedora de los elementos requeridos para definir el modelo, como la clase Model, de la cual heredarán cada clase del modelo, la clase Column para definir las columnas de las tablas, etc.

En esta aplicación es necesario implementar las clases Usuario y Comentario. La clase puede incluir el atributo de clase __tablename__ que define el nombre de la tabla en la base de datos. Flask-SQLAlchemy asigna a la tabla un nombre predeterminado si se omite __tablename__, y este nombre será el mismo que el de la clase. El resto de las variables son los atributos del modelo, definidos como instancias de la clase db.Column.

```
1  from __main__ import app  
2  from flask_sqlalchemy import SQLAlchemy  
3  
4  db = SQLAlchemy(app)  
5  
6  class Usuario(db.Model):  
7      __tablename__ = 'usuario'  
8      id = db.Column(db.Integer, primary_key=True)  
9      nombre = db.Column(db.String(80), nullable=False)  
10     correo = db.Column(db.String(120), unique=True, nullable=False)  
11     clave = db.Column(db.String(120), nullable=False)  
12     comentario = db.relationship('Comentario', backref='usuario', cascade="all, delete-orphan")  
13  
14  class Comentario(db.Model):  
15     __tablename__ = 'comentario'  
16     id = db.Column(db.Integer, primary_key=True)  
17     fecha = db.Column(db.DateTime)  
18     contenido = db.Column(db.Text)  
19     usuario_id = db.Column(db.Integer, db.ForeignKey('usuario.id'))  
20
```

En este caso la clase Usuario está relacionada con la clase Comentario. Para la implementación de dicha relación se debe tener en cuenta lo siguiente:

- El atributo id se establece como clave primaria de la tabla, ésta es administrada automáticamente por Flask-SQLAlchemy.

- b. Colocar una clave foránea en la clase de cardinalidad “muchos”, en este caso Comentario para hacer referencia al usuario que realiza el comentario.
- c. Usar el método `relationship()` del objeto SQLAlchemy en la clase de cardinalidad “1” para hacer referencia a una colección de elementos representados por la clase Comentario con la que se relaciona.
- d. Para que la relación sea bidireccional, usar el parámetro `backref` del método `relationship()`.

Paso 3

El archivo para la aplicación debe:

- Importar la clase SQLAlchemy desde el módulo `flask_sqlalchemy`.
- Crear el objeto de aplicación Flask.
- Vincular el objeto aplicación Flask con el archivo de configuración.
- Importar desde el archivo donde se definieron los modelos la instancia de la clase SQLAlchemy y las clases.

```
1 from flask import Flask, request, render_template
2 from flask_sqlalchemy import SQLAlchemy
3
4 app = Flask(__name__)
5 app.config.from_pyfile('config.py')
6
7 from models import db
8 from models import Usuario, Comentario
```

Nota: Para este caso de aplicación el archivo se guarda con el nombre **app.py**.



¿Cuál es la ventaja de establecer estos parámetros en un archivo independiente y luego vincularlo con la aplicación?

Paso 4

Definir las distintas rutas de la aplicación con la lógica necesaria para cumplir con las distintas funcionalidades y sus respectivos templates (archivos html).

```

1  from datetime import datetime
2  from flask import Flask, request, render_template
3  from flask_sqlalchemy import SQLAlchemy
4  from werkzeug.security import generate_password_hash, check_password_hash
5
6  app = Flask(__name__)
7  app.config.from_pyfile('config.py')
8
9  from models import db
10 from models import Usuario, Comentario
11
12 @app.route('/')
13 def inicio():
14     return render_template('inicio.html')
15 @app.route('/nuevo_usuario', methods = ['GET', 'POST'])
16 def nuevo_usuario():
17
18
19
20
21
22
23
24
25
26 @app.route('/nuevo_comentario', methods = ['GET', 'POST'])
27 def nuevo_comentario():
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44 @app.route('/ingresar_comentario', methods = ['GET', 'POST'])
45 def ingresar_comentario():
46
47
48
49
50
51
52
53
54
55
56 @app.route('/listar_comentarios')
57 def listar_comentarios():
58
59
60
61 @app.route('/listar_comentarios_usuario', methods = ['GET', 'POST'])
62 def listar_comentarios_usuario():
63
64
65
66
67
68
69
70
71
72
73 if __name__ == '__main__':
74     db.create_all()
75     app.run(debug = True)

```

Es fundamental la creación de la base de datos, con la instrucción `db.create_all()`, teniendo en cuenta que solo la creará si ésta no existe.

- En la aplicación que se está desarrollando la página HTML principal, llamada **inicio.html**, tiene el siguiente contenido.

```

1  <!DOCTYPE html>
2  <html lang = "en">
3  <head>
4  <title>Aplicación</title>
5  <link rel="stylesheet" href="{{ url_for('static', filename='css/estilos.css') }}">
6  </head>
7  <body>
8  <h1>Comentarios</h1>
9  <ul>
10 <li> <a href = "{{ url_for('nuevo_usuario') }}"> Registrarse como Usuario </a></li><br>
11 <li> <a href = "{{ url_for('nuevo_comentario') }}"> Comentar</a></li><br>
12 <li> <a href = "{{ url_for('listar_comentarios') }}"> Listar Todos los Comentarios </a></li> <br>
13 <li> <a href = "{{ url_for('listar_comentarios_usuario') }}"> Listar Comentarios </a></li><br>
14 </ul>
15 </body>
16 </html>

```

- Se ha definido un template base

```

1 <!DOCTYPE html>
2 <html lang="es">
3 <head>
4   <meta charset="UTF-8">
5   <title>{% block title %}{% endblock %}</title>
6   <link rel="stylesheet" href="{{ url_for('static', filename='css/estilos.css') }}">
7 </head>
8 <body>
9   {% block content %}{% endblock %}
10   <div class="myDiv">
11     <br><br><a href = "{{ url_for('inicio') }}" > Inicio </a><br><br>
12   </div>
13 </body>
14 </html>

```

- **Ruta nuevo_comentario**

El usuario debe ingresar el correo electrónico y la contraseña luego pulsar el botón “Buscar”. Si existe un usuario con los datos ingresados se despliega la página para ingresar el comentario, caso contrario se emite el error correspondiente.

La implementación de esta funcionalidad es:

```

33 @app.route('/nuevo_comentario', methods = ['GET', 'POST'])
34 def nuevo_comentario():
35     if request.method == 'POST':
36         if not request.form['email'] or not request.form['password']:
37             return render_template('error.html', error="Por favor ingrese los datos requeridos")
38         else:
39             usuario_actual= Usuario.query.filter_by(correo= request.form['email']).first()
40             if usuario_actual is None:
41                 return render_template('error.html', error="El correo no está registrado")
42             else:
43                 verificacion = check_password_hash(usuario_actual.clave, request.form['password'])
44                 if verificacion:
45                     return render_template('ingresar_comentario.html', usuario = usuario_actual)
46                 else:
47                     return render_template('error.html', error="La contraseña no es válida")
48     else:
49         return render_template('nuevo_comentario.html')
50

```

El archivo nuevo_comentario.html para ingresar correo y contraseña es el siguiente:

```

1 {% extends "base_template.html" %}
2 {% block title %}Comentar{% endblock %}
3 {% block content %}
4   <h1>Nuevo Comentario</h1>
5   <hr/>
6   <form action = "{{ request.path }}" method = "post" class= "container">
7     <label for = "email">Correo Electrónico</label><br>
8     <input type = "text" name = "email" placeholder = "email" /><br>
9     <label for = "password">Ingrese su contraseña</label><br>
10    <input type = "password" name = "password" placeholder = "password"/><br>
11
12    <br>
13    <input type = "submit" value = "Buscar" />
14  </form>
15  {% endblock %}
16

```

Para buscar el usuario con el correo y la contraseña ingresada, la aplicación realiza la siguiente instrucción:

```
usuario_actual= Usuario.query.filter_by(correo= request.form['email']).first()
```

Que accede a la tabla de usuarios para buscarlo, en caso de encontrarlo retorna el objeto.

A través del objeto query de la clase Usuario se aplica un filtro con el método `filter_by` que recibe como parámetro la condición para el filtro. Del conjunto de registros obtenidos obtiene el primero con el método `first()`.

Si se encontró un usuario con el correo ingresado en el formulario web, se verifica la contraseña a través del método `check_password_hash` que provee la clase `werkzeug.security`.

En caso de ser correcta la contraseña el flujo de ejecución continúa por la instrucción:

```
render_template('ingresar_comentario.html', usuario = usuario_actual)
```

Que redirige a la ruta **ingresar_comentario.html** con el usuario encontrado como parámetro.

Se corresponde con la siguiente implementación en la aplicación Flask

```
45 @app.route('/ingresar_comentario', methods = ['GET', 'POST'])
46 def ingresar_comentario():
47     if request.method == 'POST':
48         if not request.form['contenido']:
49             return render_template('error.html', error="Contenido no ingresado...")
50         else:
51             nuevo_comentario= Comentario(fecha=datetime.now(), contenido=request.form['contenido'], usuario_id=request.form['userId'])
52             db.session.add(nuevo_comentario)
53             db.session.commit()
54             return render_template('inicio.html')
55     return render_template('inicio.html')
```

El archivo HTML es el siguiente:

```
1 {% extends "base_template.html" %}
2 {% block title %}Nuevo Comentario{% endblock %}
3 {% block content %}
4 <body>
5     <h1>Nuevo Comentario</h1>
6     <hr/>
7     <form action = "{{ url_for('ingresar_comentario') }}" method = "post" class = "container">
8         <label for = "contenido">{{ usuario.nombre }} escribe tu comentario</label><br>
9         <textarea name = "contenido" placeholder = "Comentario" cols="60" rows="8"></textarea><br><br>
10        <input id="userId" name="userId" type="hidden" value="{{usuario.id}}">
11
12        <br>
13        <input type = "submit" value = "Enviar" />
14    </form>
15 </body>
16 {% endblock %}
```

En esta funcionalidad se debe destacar que la aplicación evalúa si el usuario ingresó texto en el componente contenido del formulario web. Si es así, crea una instancia de la clase `Comentario` a través de la siguiente instrucción:

```
nuevo_comentario= Comentario(fecha=datetime.now(), contenido =
    request.form['contenido'], usuario_id = request.form['userId'])
```

Este objeto se almacena en la base de datos a través del objeto `SQLAlchemy`, el objeto `session` y el método `add`, como se muestra a continuación:

```
db.session.add(nuevo_comentario)
db.session.commit()
```

Para que un objeto quede registrado definitivamente en la base de datos, debe confirmarse llamando al método `commit()`.

- **Ruta `listar_comentarios`**

Muestra todos los comentarios realizados.

La implementación en el archivo .py es la siguiente:

```
@app.route('/listar_comentarios')
def listar_comentarios():
    return render_template('listar_comentario.html', comentarios = Comentario.query.all())
```

Dirige a la página web pasándole como parámetro todos los comentarios, los cuales los obtiene a través de la instrucción `Comentario.query.all()`.

El HTML correspondientes es el siguiente:

```
1  {% extends "base_template.html" %}
2  {% block title %}Comentarios{% endblock %}
3  {% block content %}
4      <h1>Comentarios registrados </h1>
5      <hr/>
6      <div class="container">
7          <table>
8              <thead>
9                  <tr>
10                     <th>Usuario</th>
11                     <th>Comentario</th>
12                     <th>Fecha y hora</th>
13                 </tr>
14             </thead>
15             <tbody>
16                 {% for comentario in comentarios %}
17                     <tr>
18                         <td>{{ comentario.usuario.nombre }}</td>
19                         <td>{{ comentario.contenido }}</td>
20                         <td>{{ comentario.fecha }}</td>
21                     </tr>
22                 {% endfor %}
23             </tbody>
24         </table>
25         <br><br>
26     </div>
27 {% endblock %}
```

Es importante destacar:

- ✓ El uso de los delimitadores `{%...%}` entre el etiquetado HTML para dar dinámica al contenido de la página.
- ✓ La página recibe una colección de instancias de la clase `Comentario`, a través de la estructura iterativa `for` muestra los datos de cada comentario. Incluido el nombre del usuario el cual es atributo de la clase `Usuario` con la cual está relacionada.

- **Ruta `listar_comentarios_usuario`**

Mostrar los comentarios de un usuario seleccionado previamente.

La regla en la aplicación es la siguiente:


```

61 #app.route('/listar_comentarios_usuario', methods = ['GET', 'POST'])
62 def listar_comentarios_usuario():
63     if request.method == 'POST':
64         if not request.form['usuarios']:
65             #Pasa como parámetro todos los usuarios
66             return render_template('listar_comentario_usuario.html', usuarios = Usuario.query.all(), usuario_seleccionado = None )
67         else:
68             return render_template('listar_comentario_usuario.html', usuarios= None, usuario_selec = Usuario.query.get(request.form['usuarios']))
69     else:
70         return render_template('listar_comentario_usuario.html', usuarios = Usuario.query.all(), usuario_selec = None )
71

```

Direcciona a la página `listar_comentario_usuario.html` en dos casos. Primero, para que el usuario seleccione el nombre del usuario del cual desea conocer los comentarios; y también para mostrar los comentarios del usuario seleccionado.

El HTML correspondiente:

```

1  {% extends "base_template.html" %}
2  {% block title %}Comentario de Usuario{% endblock %}
3  {% block content %}
4  <body>
5      <h1>Comentarios registrados </h1>
6      <hr/>
7      <form action="{{ request.path }}" method="POST" class= "container">
8
9          {% if usuarios is not none %}
10             <label for = "usuarios">Usuarios</label> <br>
11             <select id= "usuarios" name="usuarios" width="500px" >
12                 {% for usuario in usuarios %}
13                     <option value="{{ usuario.id }}">{{ usuario.nombre}}</option>
14                 {% endfor %}
15             </select>
16
17             <br><br>
18
19             <input class="button1" type="submit" value="Buscar comentarios">
20         {% else %}
21             <p> {{ usuario_selec.nombre}} ({{ usuario_selec.correo}}) escribió los siguientes comentarios:</p>
22             {% for comentario in usuario_selec.comentario %}
23                 <ul>
24                     <li>{{ comentario.contenido }} </li>
25                 </ul>
26             {% endfor %}
27         {% endif %}
28     <br><br>
29 </form>
30
31 {% endblock %}

```

Evalúa que la página haya recibido un parámetro `usuarios`, si es así construye la página recorriendo la colección de usuarios que se recibió desde la aplicación Flask y generando un componente select cuyas opciones son los nombres de los usuarios registrados. Si no recibió el conjunto de usuarios, entonces va a recibir el usuario seleccionado. De dicho usuario muestra los comentarios.

- **Página de error**

Para notificar la ocurrencia de algunos errores se ha implementado la página `error.html`.

```

1  {% extends "base_template.html" %}
2  {% block title %}Error{% endblock %}
3  {% block content %}
4      <h1> {{ error }} </h1>
5  {% endblock %}

```

Que también hace uso de la plantilla base.

Bibliografía

Flask Web Development Developing Web Applications with Python. Miguel Grinberg. O'Reilly Media, segunda edición. 2018.

Aprendizaje Flask. Ebook. <https://riptutorial.com/Download/flask-es.pdf>

Enlaces de interés

<https://flask.palletsprojects.com/en/1.1.x/>

<https://flask.palletsprojects.com/en/2.0.x/>

<https://flask.palletsprojects.com/en/2.0.x/api/>

<http://www.manualweb.net/flask/mi-primer-programa-flask/>

<https://www.tutorialspoint.com/flask/index.htm>

<https://docs.sqlalchemy.org/en/13/orm/tutorial.html>

Anexos

Conceptos de redes

- **Cliente-Servidor**

La arquitectura cliente-servidor es un modelo de aplicación en el que las tareas se reparten entre los proveedores de recursos o servicios, llamados **servidores**, y los demandantes, llamados **clientes**. Un cliente realiza peticiones al servidor quien le da respuesta.

- **Generación dinámica de páginas web**

El servidor ejecuta software que genera automáticamente archivos html que son visualizados en el navegador del cliente. El cual es independiente de la tecnología que esté usando el servidor.

- **HTML (Hyper Text Markup Language - Lenguaje de marcación de hipertexto)**

HTML es el lenguaje que se emplea para el desarrollo de páginas web. Es usado para describir la estructura y el contenido en forma de texto, así como para complementar el texto con objetos tales como imágenes. El HTML se escribe en forma de «*etiquetas*», rodeadas por corchetes angulares (<,>), las cuales, son interpretadas por el navegador y desplegadas en la pantalla con aspecto gráfico. Existen etiquetas para imágenes, hipervínculos, listas, tablas para tabular datos, etc.

Para crear una página HTML se requiere un simple editor de texto. Este tipo de archivo es estático, es decir que mostrarán siempre el mismo contenido.

- **Hypertext Transfer Protocol o HTTP**

HTTP (protocolo de transferencia de hipertexto) es el protocolo usado en cada transacción de la World Wide Web. Este protocolo define la sintaxis y la semántica que utilizan los elementos de software de la arquitectura web (clientes, servidores, proxies) para comunicarse. Es un protocolo orientado a transacciones y sigue el esquema petición-respuesta entre un cliente y un servidor. Al cliente que efectúa la petición (un navegador web) se lo conoce como agente del usuario. A la información transmitida se la llama recurso y se la identifica mediante un localizador uniforme de recursos (URL).

- **Navegador**

Es una aplicación cliente que utiliza el protocolo http para acceder a máquina donde están alojados los archivos.

- **Protocolo de comunicaciones**

Es el conjunto de reglas normalizadas para la representación, señalización, autenticación y detección de errores necesario para enviar información a través de un canal de comunicación.

- **Servidor web**

Un servidor web o servidor HTTP es un programa informático que procesa una aplicación del lado del servidor realizando conexiones con el cliente, generando respuestas en cualquier lenguaje o aplicación del lado del cliente. El código recibido por el cliente es ejecutado por un navegador web. Para estas comunicaciones se utiliza el protocolo HTTP. El término también se emplea para referirse a la computadora que ejecuta el programa.

Conceptos de Base de Datos

- Una *base de datos* es un conjunto de tablas.
- Una *tabla* es un conjunto de filas.
- Cada *fila* es un registro.
- Cada columna es un *campo de datos* de un registro.

El diseño de las columnas, en términos de nombre, tipo de datos y distintos atributos, como valores predeterminados y si una columna puede o no ser NULL (quedar en blanco), se conoce como *esquema*.

Ejemplo: Una base de datos de una pinacoteca puede contener las tablas: Obras, Autores.

- En la tabla Obras cada fila representa una obra pictórica de algún autor.
- Las columnas representan los siguientes campos: ID_Obra, Denominación, ID_Autor, Fecha.
- En la tabla Autores cada fila representa un artista plástico.
- Las columnas representan los siguientes campos: ID_Autor, Nombre, Pais.