Imagen que contiene Texto

Descripción generada automáticamente

**INSTITUTO TECNOLÓGICO DE ESTUDIOS**

**SUPERIORES DE MONTERREY**

**Actividad Individual 2**

**Materia**

Diseño e implementación de sistemas mecatrónicos

**Ingenieros**

Cesar Emilio Castaño Marín - A00830006

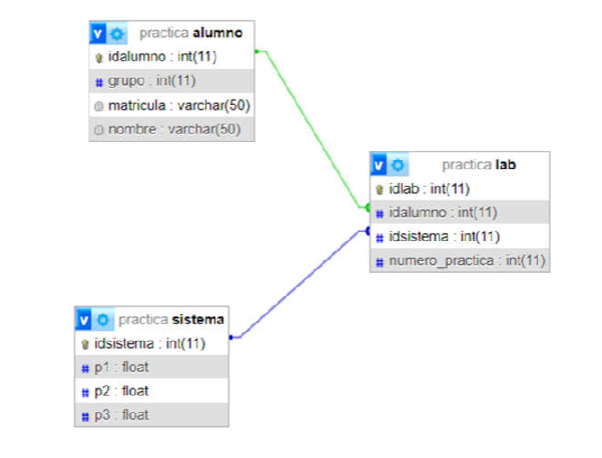
 **Profesor**

 Valery Moreno

26 de Mayo de 2024

**Situación Problema**

El siguiente esquema muestra la base de datos practica ligeramente modificada con relación a la vista en clases. Se ha añadido el campo *nombre* en la tabla alumno y el campo *numero\_practica* en la tabla lab:

  
*Figura. Estructura solicitada de la dB*

Se requiere realizar un sitio web que permita (a cada punto le llamaremos petición):

* Dar de alta nuevos alumnos (agregarlo en la tabla correspondiente).
* Dar de alta un nuevo sistema (agregarlo en la tabla correspondiente).
* Modificar un alumno (actualizarlo en la tabla correspondiente).
* Modificar un sistema (actualizarlo en la tabla correspondiente).
* Dar de alta una nueva práctica (agregarla en la tabla correspondiente).
* Listar los alumnos y sistema dado un número de práctica.

**Diseño de la Base de Datos**

A partir de la estructura de dB solicitada, se hizo uso de Heidi para crear una base de datos en local que contara con aquella estructura. A continuación, se muestran las 3 tablas creadas:

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente  
*Figura. Tabla alumno en dB en local*

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente  
*Figura. Tabla sistema en dB en local*

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente  
*Figura. Tabla lab en dB en local con las llaves foráneas apuntando al id del alumno y sistema*

Como se puede ver en las imágenes anteriores, las tablas junto con los tipos de dato de cada campo fueron creadas a medida con la estructura solicitada. Estas tablas no poseen relaciones complejas, sin embargo, la tabla lab cuenta con dos llaves foráneas que corresponden al *idsistema* y *idalumno*, provenientes de la tabla de sistema y alumno respectivamente.

Las relaciones en este caso son de muchos a muchos, ya que tanto un alumno como un sistema, pueden estar en muchas prácticas, y a la vez, una práctica puede tener muchos alumnos y sistemas.

**Estructura de Repositorio**

**Texto

Descripción generada automáticamente**

*Figura. Repositorio con archivos para desarrollo de actividad*

En este caso podemos ver que se ha utilizado una estructura común para un directorio para desarrollo web. En la carpeta static, se han colocado el css y el app de javascript (para dar dinamismo al sitio web). En la carpeta de templates se han colocado todas las plantillas de html; index.html es el menú principal del sitio web (este será explicado más adelante), y los demás HTMLs corresponden a páginas diferentes para cumplir cada función solicitada.

Por último, el backend en Python se encuentra en la carpeta principal, al mismo nivel que las carpetas de static y templates.

**Link al Repositorio en Github:**

**Plantillas HTML**

Como ya se mencionó, se ha construido un HTML individual para cada funcionalidad solicitada. Con el fin de no saturar el reporte, y debido a que todas las plantillas cuentan con una estructura muy similar, se explicarán únicamente el menú principal (index.html), y la página para dar de alta un alumno (alta\_alumno.html).

Texto

Descripción generada automáticamente  
*Figura. Código HTML de index.html*

En la plantilla anterior para el menú principal, podemos ver una estructura simple de un HTML. Primeramente, definimos el tipo de documento como HTML con *doctype*, luego lo inicializamos con *html* y definimos el idioma del contenido a español (*lang = “es”*).

Luego, en la cabecera hacemos uso de *meta charset* para especificar la codificación de caracteres como UTF-8, lo cual es estándar para soportar caracteres internacionales. Después configuramos la ventana para que el diseño sea receptivo en cualquier dispositivo. *width=device-width* ajusta el ancho de la página al ancho del dispositivo e *initial-scale=1.0* establece el nivel de zoom inicial. Aquí también se define el título de la página que se muestra en la pestaña del navegador, y se hace uso de *<link rel="stylesheet" href="{{ url\_for('static', filename='styles.css') }}">* para vincular la hoja de estilos CSS. La función anterior es una plantilla de **Flask** que genera la URL para el archivo CSS ubicado en la carpeta static.

Se hace necesario un paréntesis en esta sección para dar una introducción a **Flask**. Este es un microframework para el desarrollo de aplicaciones web en Python. Fue diseñado para ser fácil de usar y flexible, es por esto que utiliza un sistema de rutas sencillo y claro para mapear URLs a funciones Python. Esto permite definir fácilmente qué código se ejecuta cuando un usuario accede a una URL específica (el funcionamiento de este framework será explicado en la sección de backend de este documento).

Ahora, regresando al HTML, después de la cabecera se construye el cuerpo de la página, aquí se hace uso de *h1* para definir el título principal de la página, se hace uso de *div* e *image* para colocar la imagen de Cesar (su servidor) en un contenedor debajo del título, y se hace uso de otro contenedor para almacenar todos los botones que tienen direccionamientos hacia las páginas del sitio web. Se usa la *a* en HTML para establecer referencias, y en este caso se vuelve a hacer uso de la sintaxis relacionada con Flask.

En el contexto de una aplicación Flask, *url\_for('ruta')* es una función que genera una URL para la función de vista dada. Esto permite cambiar las rutas de la aplicación sin tener que actualizar los enlaces manualmente en las plantillas HTML. Por ejemplo, si la ruta alta\_*alumno* está definida en Flask, *url\_for('alta\_alumno')* generará la URL correspondiente a esa ruta.

Texto

Descripción generada automáticamente  
*Figura. Código HTML de alta\_alumno.html*

Ahora, el HTML para la página de dar de alta un alumno tiene la misma estructura que index.html; la diferencia en este caso corresponde al formulario que contiene esta nueva página. Podemos ver que dentro del cuerpo de la página se incluye un formulario con id “*alta-alumno-form*”, este formulario contiene los inputs para crear un alumno: grupo, nombre y matrícula. Cada input se le asigna un tipo predeterminado y se le da la característica de requerido, de esta forma se limita lo que el usuario puede ingresar en los campos del formulario.

El formulario contiene también un botón de tipo *submit*, que será el evento que se escuchará para captar la información del formulario en la app de javascript. Después del formulario podemos ver que la página cuenta con un botón para regresar al menú principal, y una referencia al *script* que se comunicará con la página web, en este caso el app.js.

**Estilos con CSS**

Para el estilo del sitio web se hizo uso de una plantilla CSS básica, esta fue referenciada en todas las páginas del sitio web, y se crearon algunas clases específicas para algunos objetos de las páginas (como los son la clase “*button*” o “*circular-image*”).

Texto

Descripción generada automáticamente

*Figura. Código CSS de body, h1 y form*

En la figura anterior se puede apreciar una configuración básica de algunos elementos del sitio web. Vemos que se definen colores de fondo, alineaciones, anchos, márgenes, etc. En esta figura se muestra únicamente el estilo para el body, h1 y form de todo el sitio web, sin embargo, el archivo CSS cuenta con los estilos para cada sección de las páginas.

Texto

Descripción generada automáticamente  
*Figura. Estilo CSS para la clase button y circular-image*

En la figura anterior se puede apreciar el estilo para las clases que se crearon tanto para los contenedores, como para los objetos con clase *button* y la imagen de la página principal. Se puede apreciar que se definieron los mismos parámetros de color de fondo, márgenes, etc; también se redimensionó la imagen y se le dio un radio al borde para que adquiriese la forma de un círculo. Para los botones se le dio un pequeño cambio de color para cuando se coloca el mouse encima de los mismos (*hover*).

**Dinamismo Haciendo Uso de Javascript**

Como ya se mencionó, para dar dinamismo al sitio web y comunicar el frontend con el backend, se hizo uso de una aplicación en javascript. La app de javascript contiene diferentes funciones que hacen uso de los métodos http ‘POST’ y ‘GET’ para enviar y solicitar datos del servidor.

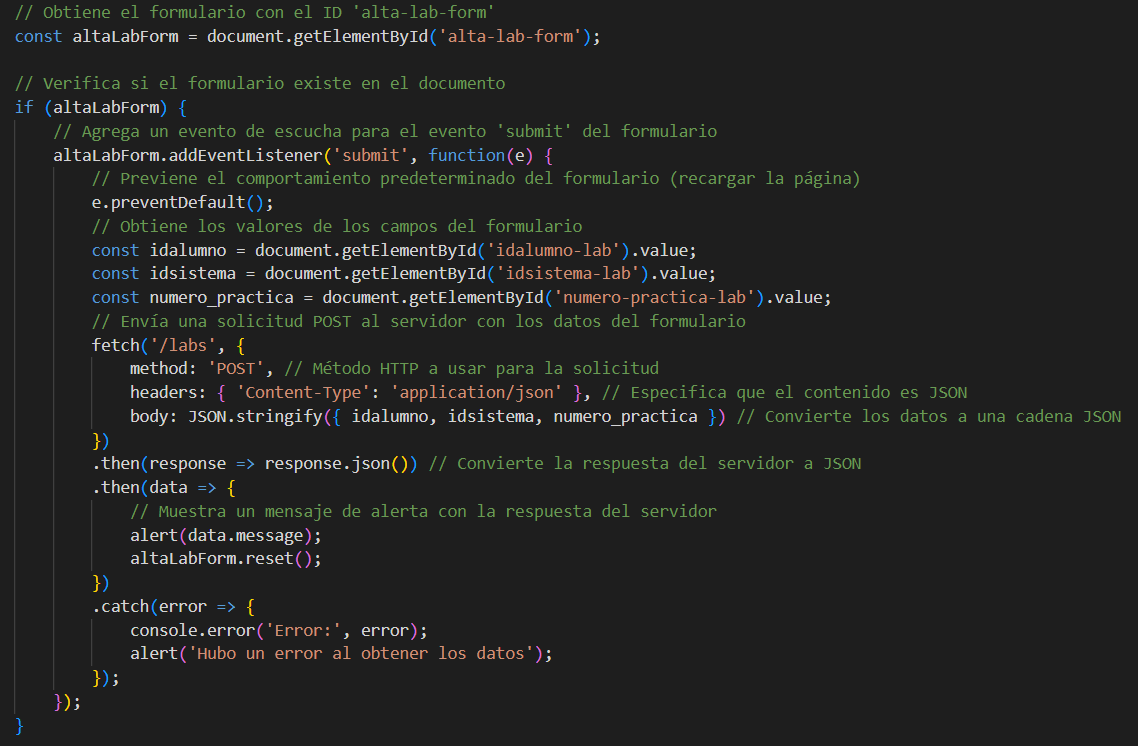
En este caso, las funcionalidades que causan cambios en el servidor (como lo son registros y actualizaciones en la dB) y por tanto hacen uso del método ‘POST’ corresponden a:

* Insertar alumno
* Insertar sistema
* Actualizar alumno
* Actualizar sistema
* Insertar práctica

En este caso se explicará únicamente la función de insertar práctica, ya que todas las funciones poseen la misma estructura e implementan el mismo método ‘POST’ como se mencionó anteriormente. Antes de explicar este código, hay que recordar que la app.js está siendo referenciada como script en todos los html (menos en el index.html), y es de esta forma que se hace posible que la app escuche los eventos que ocurren en el frontend.

  
*Figura. Evento de escucha inicial del app en javascript*

La app de JS comienza con una acción de escucha hacia el evento *DOMContentLoaded*, el cual es un evento del Document Object Model (interfaz de programación para documentos HTML) que se dispara cuando el documento HTML inicial ha sido completamente cargado y analizado.

  
*Figura. Primera parte de la función con método POST para enviar información de una nueva práctica al servidor*

Una vez que el HTML ha cargado, hay diferentes constantes que se están creando, todas corresponden a los diferentes formularios de las páginas del sitio web. En la figura anterior se puede apreciar que en este caso se está obteniendo el formulario *altaLabForm* mediante el id que este formulario posee en el HTML (‘*alta-lab-form*’).

Si el objeto sí pudo ser obtenido (o sea que la página que se cargó sí contenía el formulario con el id indicado), se crea una acción de escucha para el evento de tipo *submit* proveniente del frontend, o sea cuando se presiona el botón de enviar formulario y efectivamente se envía la información de este para dar de alta una nueva práctica.

Dentro del manejador del evento de tipo *submit*, se llama la función *preventDefault*, que nos permite tener un control total sobre lo que sucede cuando se dispara un evento. Para el caso de formularios, la acción predeterminada a realizar cuando se acciona el evento de tipo *submit* (se envía el formulario) corresponde a recargar la página en su totalidad; en nuestro caso, evitamos que pase esto con la función mencionada y simplemente reiniciamos el formulario una vez que culmina la función.

Una vez que se ha prevenido la recarga de la página, se obtienen y almacenan los inputs del formulario en cuestión en nuevas constantes, esto mediante la obtención del valor de los espacios de input referenciados con su id. Estos valores son los que se envían al servidor por medio de un JSON, dentro del cuerpo de la solicitud *fetch* que se realiza en la ruta *‘/labs’* mediante el método ‘POST’.

Texto

Descripción generada automáticamente  
*Figura. Segunda parte de la función con método POST para enviar información de una nueva práctica al servidor*

Una vez que se ha realizado la solicitud *fetch* y se ha recibido una respuesta del servidor, se convierte la respuesta a una estructura JSON, se genera una alerta con el mensaje proveniente del servidor y se reinicia el formulario para dar de alta una práctica. En caso de que haya ocurrido algún error durante el proceso, el error será capturado por el código y mostrado en una alerta que se desplegará sobre la página web.

Ahora veremos la única función en la que se hace uso del método ‘GET’.

Texto

Descripción generada automáticamente  
*Figura. Sección de código de la función para listar un número de práctica*

La funcionalidad de listar los alumnos con sus sistemas con un mismo número de práctica es la única función donde se implementó el método ‘GET’. En este caso no se hace necesario realizar cambios en el servidor, solamente se solicita la información de las prácticas pasando el número de práctica a consultar como un parámetro en la URL o el enrutamiento (en este caso *`/labsListar?numero\_practica=${numero\_practica}*`).

Otra funcionalidad extra relacionada con esta función corresponde a la inserción de los datos encontrados en el HTML. Como se puede ver en el código, se obtiene el objeto *result* mediante su id (este es un contenedor en el HTML de la página para listar una práctica) y se agrega la nueva información a su interior mediante la propiedad *innerHTML*.

**Backend (Servidor) con Python**

Ahora, el backend fue programado con Python haciendo uso de la librería Flask para establecer la conexión con el servidor (en este caso se hace uso del servidor local con el puerto 5000, predeterminado para Flask), y de la librería mysql-connector para establecer la conexión y hacer las consultas a la dB.

Texto

Descripción generada automáticamente  
*Figura. Función para crear la conexión con la dB haciendo uso de la librería mysql-connector*

El código se divide en tres partes: la primera corresponde a la función para crear la conexión con la dB (se puede apreciar en la figura anterior), la segunda corresponde a los enrutamientos primarios que simplemente renderizan las plantillas HTML, y la última parte corresponde a las funciones que hacen los request a la dB y que escuchan los enrutamientos donde se están realizando los request http ‘POST’ y ‘GET’ desde el cliente, o sea la app.js.

**Texto

Descripción generada automáticamente***Figura. Enrutamientos primarios para renderizar páginas con Flask*

En la figura anterior se puede apreciar que se importa Flask y se inicializa una instancia de este framework con *Flask(\_\_name\_\_)*. Esta instancia será la aplicación WSGI (Web Server Gateway Interface) que manejará las solicitudes y respuestas de la aplicación web. El parámetro *\_\_name\_\_* se pasa a la instancia de Flask para ayudar a la aplicación a determinar la ubicación del archivo principal del programa, lo cual es útil para encontrar recursos estáticos y plantillas.

También se pueden apreciar algunas funciones con enrutamientos primarios, que lo único que hacen es renderizar las plantillas HTML correspondientes a la ruta definida (estas rutas son las que se llaman en el html mediante *url\_for*).

Nuevamente y por fines prácticos, se explicará únicamente una de las funciones de la tercera parte del código. Todas las funciones tienen una estructura muy similar: recepción de datos, definición de parámetros de consulta a la dB, consultas a la dB, retorno de mensajes y/o valores.

Para dar continuidad a la explicación anterior de la app.js, se procederá entonces a explicar la función para listar una práctica, que se encuentra enrutada en *‘/labsListar’*.

**Texto

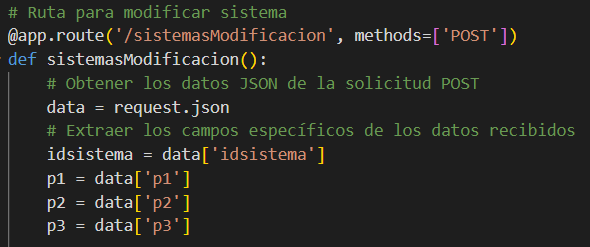
Descripción generada automáticamente***Figura. Función para listar práctica en el backend*

Como se puede apreciar en la figura anterior, se establece un decorador con los parámetros de la ruta y el método http (en este caso ‘GET’) para la función *labsListar*. Como se ha utilizado el método ‘GET’, se hace uso del método *get* del objeto *request.args* para obtener el parámetro *numero\_practica* que viene incluido en la URL o el enrutamiento proveniente de la app.js.

Una vez que se tiene el parámetro, se establece la conexión con la dB y se empiezan a realizar los request a la misma. En este caso se comienza por hacer el intento de seleccionar la práctica con el número solicitado, si la práctica existía (o sea si los resultados de la selección no están vacíos) se continúa ejecutando la función, sino se cierra la conexión con la dB y se retorna un mensaje de que no existe el número de práctica solicitado.

Si la práctica existe, se procede a seleccionar toda la información de los alumnos y sistemas que tienen registros en la dB para la práctica con el número solicitad. Se hace uso de la función fetchall para capturar todos los datos encontrados y posteriormente se crea una lista con todos los registros encontrados en formato de diccionario. Finalmente se cierra la conexión con la dB y se retorna un mensaje de que se ha listado la práctica correctamente junto con la lista de alumnos con sus sistemas correspondientes.

De las otras funciones solo cabe resaltar que el método que se define corresponde a ‘POST’, y los datos que se reciben son transformados a una estructura JSON para posteriormente obtener los parámetros, tal y como se ve a continuación.

  
*Figura. Sección inicial de una función que hace uso del método ‘POST’*

Por último, solo cabe aclarar que para ejecutar la aplicación Flask, se añade un bloque condicional al final del archivo:

  
*Figura. Condicional para ejecutar aplicación de Flask*

Si el módulo actual es el módulo principal (es decir, el archivo que se está ejecutando directamente), *\_\_name\_\_* será igual a *"\_\_main\_\_".* Esto permite que Flask configure las rutas y los recursos correctamente. *app.run(debug=True)* ejecuta la aplicación en modo de depuración, lo cual es útil durante el desarrollo ya que permite ver errores detallados y reiniciar automáticamente el servidor cuando se realizan cambios en el código.

**Resultados de Funcionamiento y Pantallas**

Ya que se han explicado los códigos, se deja un video corto que muestra el funcionamiento correcto del sitio web. En este video se pueden ver todas las pantallas junto con las consultas a la dB funcionando correctamente.

**Video:** [**https://youtu.be/3bBsM81mER0**](https://youtu.be/3bBsM81mER0)