



Universidad Nacional de La Plata

Facultad de Informática

Taller de Proyecto II

Práctica 1: Simulación

Cobanera, Santiago

Pontetto, Axel

14 de septiembre de 2017

Ejercicio 1:

Generar el archivo 'requirements.txt' con las dependencias necesarias para poder levantar un servidor con Flask. Explicar un ejemplo de uso con la secuencia de acciones y procesos involucrados desde el inicio de la interacción con un usuario hasta que el usuario recibe la respuesta.

Instalación de Flask y sus dependencias

Para poder levantar un servidor Flask en python necesitamos tener instalado python en el equipo a usar, así como también, el administrador de paquetes de python PIP. En este informe se da por entendido que python está instalado en su equipo, en caso contrario, podemos descargarlo desde su sitio web oficial, <https://www.python.org/>, y una vez instalado podemos descargar el administrador de paquetes de python, PIP, también desde su sitio web oficial, <https://pypi.python.org/pypi/pip>.

Una vez instalado las herramientas necesarias en el equipo, procedemos a crear un proyecto para usar el micro framework Flask. Para esto creamos un directorio con el nombre, en este ejemplo, Ejercicio_1. Dentro de dicho directorio creamos un nuevo directorio llamado templates para, posteriormente, generar allí las vistas html correspondientes a la aplicación web. Seguido de esto, a la altura del directorio templates, creamos un archivo vacío llamado requirements.txt, el cual, contendrá las dependencias necesarias para la instalación del micro framework Flask. En la siguiente imagen podemos ver la estructura de directorios de lo expuesto hasta el momento.

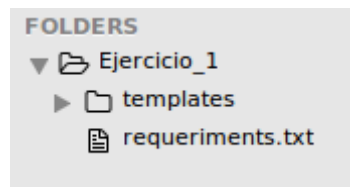


Figura 1

Ahora procedemos a instalar dicho micro framework Flask. Para lograr esto, cargamos en el archivo requirements.txt, creado anteriormente, básicamente, dos paquetes:

- **wheel:** este paquete no es ni más ni menos que otro administrador de paquetes como PIP. Esta dependencia debe ser instalada debido a que Fask utiliza este como parte de su propia instalación.
- **Flask:** como su nombre lo infiere, dicha dependencia instalará el micro framework en cuestión.

A Continuación, se observa como quedo cargado el archivo requirements.txt.

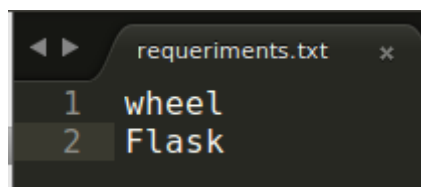


Figura 2

Como es de intuir, la simple carga del archivo requirements.txt no carga por si solo el micro framework Flask y sus dependencias. Para aplicar estos cambios ingresados debemos escribir, en la consola de su sistema operativo favorito, simplemente, pip install -r requirements.txt. Este comando le indica al administrador de paquetes PIP que los paquetes y sus dependencias a instalar se encuentran en el archivo antes mencionado. Es importante mencionar que el comando pip install -r requirements.txt debe ser ejecutado desde donde esté ubicado dicho archivo, pues, caso contrario, fallará debido a que no puede ser encontrado. En la siguiente imagen podemos ver la secuencia anterior.

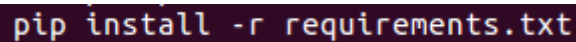


Figura 3

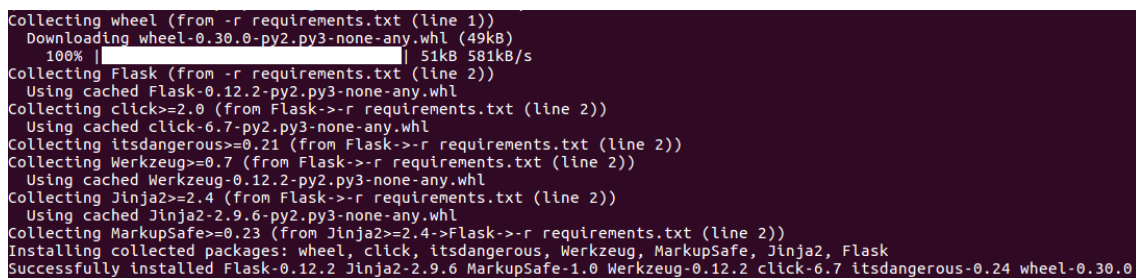


Figura 4

Aplicación de ejemplo y flujo de la aplicación web

Ahora que ya disponemos de todas las herramientas necesarias, procedemos a la creación de una aplicación web sencilla ejemplificando el flujo de dicha aplicación, es decir, desde que el usuario ingresa con su navegador y recibe una respuesta. Primero, a la altura del archivo requirements.txt, creamos un nuevo archivo llamado app.py y lo cargamos con el código Python detallado a continuación.

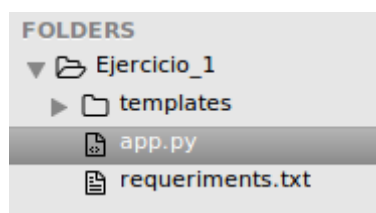


Figura 5

```

1 # Librerías para importar en el proyecto
2 from flask import Flask
3 from flask import render_template
4 from flask import request
5
6 app = Flask(__name__)
7
8 # Definimos la ruta destinada para ingresar con un navegador web
9 # En este caso definimos la ruta /
10 # Request por defecto: GET
11 @app.route('/')
12 # Función destinada para controlar la ruta / definida anteriormente
13 def index():
14     # Retornamos el template formulario ubicado en la carpeta templates
15     return render_template('formulario.html')
16
17 if __name__ == "__main__":
18     # Definimos el puerto para correr la aplicación
19     # Por defecto corre en localhost
20     app.run(debug = True, port = 8000)

```

Figura 6

En dicho archivo definimos que la ruta raíz de la aplicación web, es decir el "/", recibirá una petición, generada por un usuario al acceder al sitio web con su navegador, http con el verbo GET mediante `@app.route('/')`. Dicha petición será manejada con la definición de la función `def index()`, la cual retorna un template llamado `formulario.html`. Dicho template, simplemente mostrará al usuario un formulario para que complete con su nombre y apellido. Para esto creamos, dentro del directorio `templates`, el archivo `formulario.html` con el código que se visualiza en las siguientes imágenes.

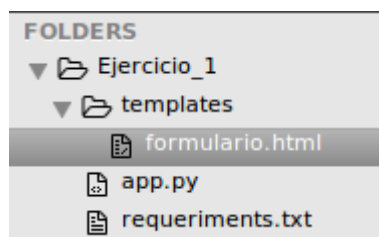


Figura 7

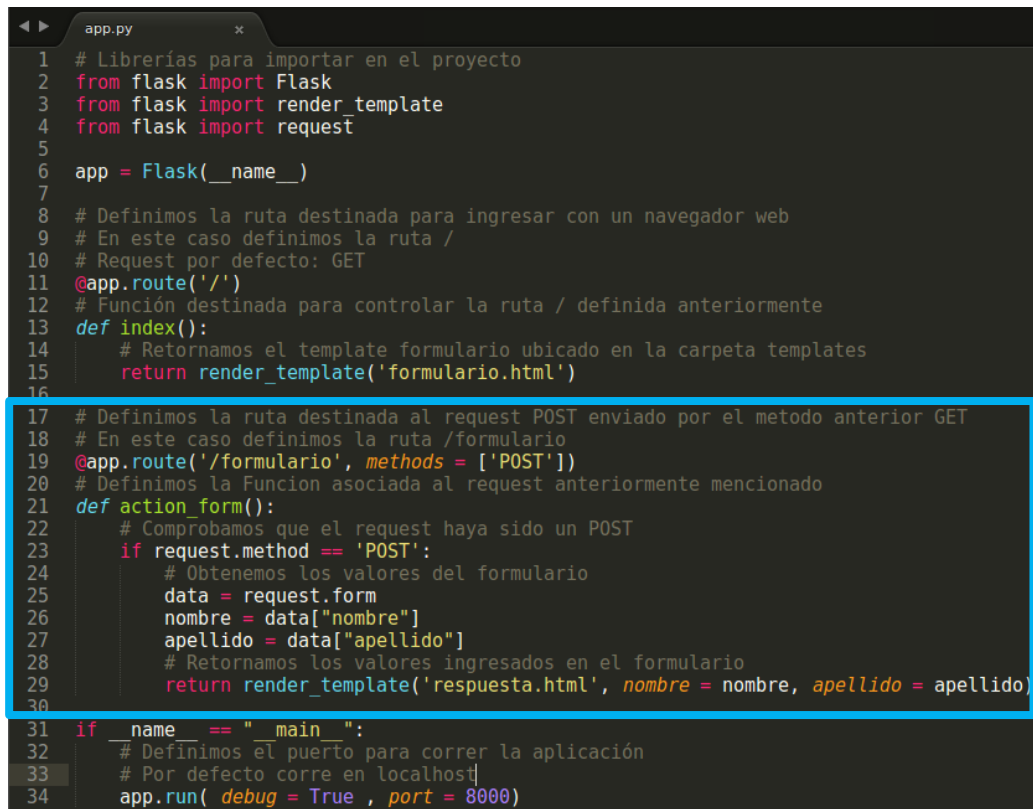
```

1 <html>
2 <head>
3 <title>Trabajo Practico Número Uno</title>
4 </head>
5 <body>
6
7 <h1>Formulario de Ejemplo</h1>
8
9 <!-- Elemento HTML para definir un formulario -->
10 <form action="/formulario" method="POST">
11 <!-- Elemento HTML para definir un etiqueta -->
12 <label for="nombre">Nombre: </label> <br> <br>
13 <!-- Elemento HTML para definir un campo de texto -->
14 <input type="text"
15       id="nombre"
16       name="nombre"
17       placeholder="Ingrese su nombre" required/> <br> <br>
18 <!-- Elemento HTML para definir un etiqueta -->
19 <label for="apellido">Apellido: </label> <br> <br>
20 <!-- Elemento HTML para definir un campo de texto -->
21 <input type="text"
22       id="apellido"
23       name="apellido"
24       placeholder="Ingrese su Apellido" required/> <br> <br>
25 <!-- Elemento HTML para definir un boton submit -->
26 <button type="submit">Sumírse en Flask</button>
27
28 </form>
29
30 </body>
31 </html>

```

Figura 8

El formulario anterior consta básicamente de dos inputs html, uno para el ingreso del nombre y otro para el ingreso del apellido, los cuales, se encuentran dentro de un formulario html. Además, se crea un botón del tipo submit, que cuando es clickeado por el usuario, genera un requerimiento http del tipo POST a la dirección /formulario, enviando, además, los valores ingresados en los inputs mencionados. Para recibir este pedido http, debemos volver al archivo app.py, generado anteriormente, y añadir el código necesario para definir la ruta deseada junto con su respectiva función para el manejo de dicho requerimiento POST.



```
1 # Librerías para importar en el proyecto
2 from flask import Flask
3 from flask import render_template
4 from flask import request
5
6 app = Flask(__name__)
7
8 # Definimos la ruta destinada para ingresar con un navegador web
9 # En este caso definimos la ruta /
10 # Request por defecto: GET
11 @app.route('/')
12 # Función destinada para controlar la ruta / definida anteriormente
13 def index():
14     # Retornamos el template formulario ubicado en la carpeta templates
15     return render_template('formulario.html')
16
17 # Definimos la ruta destinada al request POST enviado por el metodo anterior GET
18 # En este caso definimos la ruta /formulario
19 @app.route('/formulario', methods = ['POST'])
20 # Definimos la Funcion asociada al request anteriormente mencionado
21 def action_form():
22     # Comprobamos que el request haya sido un POST
23     if request.method == 'POST':
24         # Obtenemos los valores del formulario
25         data = request.form
26         nombre = data["nombre"]
27         apellido = data["apellido"]
28         # Retornamos los valores ingresados en el formulario
29         return render_template('respuesta.html', nombre = nombre, apellido = apellido)
30
31 if __name__ == "__main__":
32     # Definimos el puerto para correr la aplicación
33     # Por defecto corre en localhost
34     app.run( debug = True , port = 8000)
```

Figura 9

Como puede observarse en el código anterior, la función `def action_form()`, que atiende el pedido POST hacia la dirección /formulario, comprueba que se trate, efectivamente, de un requerimiento POST, obtiene los valores ingresados al formulario y devuelve los mismos a un nuevo template llamado `respuesta.html`. Por último, para completar el flujo, debemos crear el archivo `respuesta.html`, en el directorio `templates`, y llenarlo con el código mostrado a continuación, el cual permite, que el usuario visualice en su navegador el nombre y apellido ingresados en el formulario que fue enviado por dicho usuario.

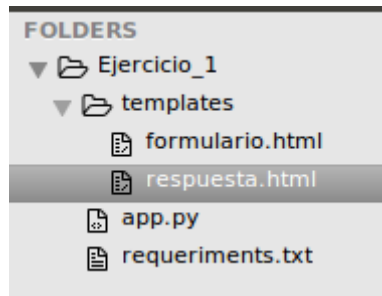


Figura 10

```
respuesta.html x
1 <html>
2   <head>
3
4     <title>Trabajo Practico N&uacute;mero Uno</title>
5
6   </head>
7
8   <body>
9
10    <h1>Usted se ha sumergido en Flask exitosamente</h1>
11    <!-- Imprimimos el nombre en el navegador -->
12    <h3>Nombre: {{ nombre }} </h3>
13    <!-- Imprimimos el apellido en el navegador -->
14    <h3>Apellido: {{ apellido }} </h3>
15
16  </body>
17 </html>
```

Figura 11

Por último, solo nos resta ejecutar la aplicación desarrollada. Esto lo realizamos directamente desde la consola del sistema operativo ejecutando la orden `python app.py`. Cabe mencionar que la ejecución de esta orden se debe realizar desde el directorio en donde se encuentra la aplicación, en este caso `Ejercicio_1`.

```
* Running on http://127.0.0.1:8000/ (Press CTRL+C to quit)
* Restarting with stat
* Debugger is active!
* Debugger PIN: 124-000-322
```

Figura 12

Resumen: generalización del flujo de la aplicación

De lo expuesto anteriormente, puede generalizarse la secuencia de acciones que ocurre desde que el usuario realiza una interacción con la aplicación hasta que el mismo recibe la respuesta. Dicha secuencia para la aplicación de ejemplo es la que sigue a continuación.

1. Usuario ingresa a su navegador de preferencia y escribe la dirección <http://127.0.0.1:8000/>, el cual envía un requerimiento http get a la dirección “/”.

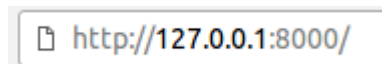


Figura 13

2. El requerimiento anterior es atendido por la función `def index()` y retorna el template `formulario.html`

Formulario de Ejemplo

Nombre:

Apellido:

Figura 14

3. El usuario visualizar el formulario anterior y rellena las entradas de nombre y apellido, además de realizar el envía del mismo por medio del botón: Sumérjase en Flask

Formulario de Ejemplo

Nombre:

Apellido:

Figura 15

4. El envío del formulario por parte del usuario genera un requerimiento http del tipo POST a la dirección `"/formulario"`, el cual es manejada por la función `def action_form()`

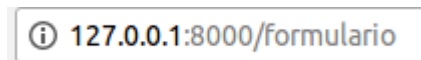


Figura 16

5. La función anterior obtiene los datos del nombre y apellido y retorna el template `respuesta.html`, mostrando dichos datos

Usted se ha sumergido en Flask exitosamente

Nombre: Bill

Apellido: Jobs

Figura 17

Ejercicio 2

Desarrollar un experimento que muestre si el servidor HTTP agrega o quita información a la genera un programa Python. Nota: debería programar o utilizar un programa Python para conocer exactamente lo que genera y debería mostrar la información que llega del lado del cliente, a nivel de HTTP o, al menos, a nivel de HTML (preferentemente HTTP).

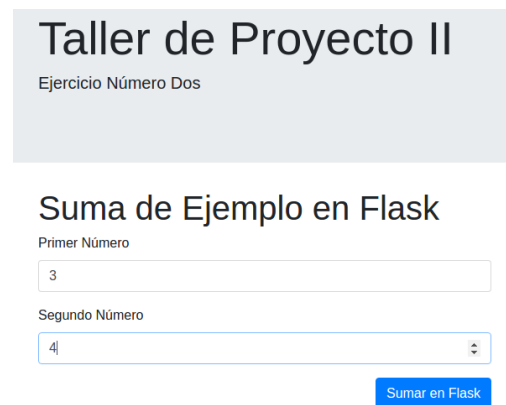
Sumador de Números enteros en Flask

Para la resolución de este ejercicio se propuso realizar una aplicación web que realice la suma de dos números enteros ingresados por el usuario mediante su navegador web de preferencia. Como el objetivo de este ejercicio es visualizar si se agrega información adicional a una vista HTML y al protocolo HTTP, en comparación con los códigos correspondientes de la aplicación desarrollada, no se explicará en detalle la implementación de dichos códigos, sino que, se analizará por separado tanto las vistas HTML generadas como el protocolo de aplicación HTTP. Para mayor detalle de la implementación el código puede visualizarse en el anexo de códigos. A continuación se puede visualizar la mencionada aplicación.



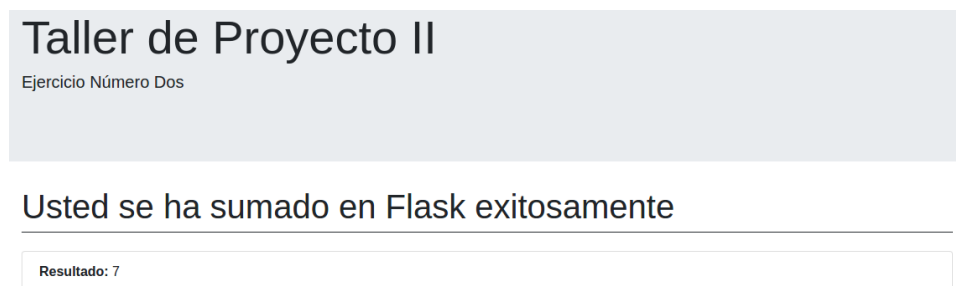
The screenshot shows a web form titled "Taller de Proyecto II" with the subtitle "Ejercicio Número Dos". Below the title is the heading "Suma de Ejemplo en Flask". There are two input fields: "Primer Número" with the placeholder text "Ingrese el Primer Número" and "Segundo Número" with the placeholder text "Ingrese el Segundo Número". A blue button labeled "Sumar en Flask" is positioned at the bottom right of the form.

Figura 18



The screenshot shows the same web form as Figure 18, but with the values "3" and "4" entered into the "Primer Número" and "Segundo Número" fields respectively. The blue "Sumar en Flask" button remains at the bottom right.

Figura 19



The screenshot shows the result of the sum operation. The heading "Suma de Ejemplo en Flask" is replaced by the message "Usted se ha sumado en Flask exitosamente". Below this message is a white box with the text "Resultado: 7".

Figura 20

Análisis de la generación de código HTML

En el correspondiente desarrollo de esta aplicación web, se decidió usar el framework front end, ampliamente utilizado, llamado Bootstrap. De este modo, se puede apreciar

mejor las diferencias entre el código que nosotros programamos y el visualizado finalmente por el usuario.

Antes de realizar la comparación en cuestión, definamos primero qué es Bootstrap. Bootstrap, es un framework originalmente creado por Twitter, que permite crear interfaces web con CSS y JavaScript, cuya particularidad es la de adaptar la interfaz del sitio web al tamaño del dispositivo en que se visualice. Es decir, el sitio web se adapta automáticamente al tamaño de una PC, una Tablet u otro dispositivo. Esta técnica de diseño y desarrollo se conoce como “responsive design” o diseño adaptativo. Por lo tanto, nosotros podemos utilizar ciertas clases de estilos y funcionalidades ya predefinidas por dicho framework, de forma tal, de realizar diseños elegantes con mucho menos esfuerzos de codificación.

- **Comparación del código HTML desarrollado en la aplicación web respecto del generado en el navegador**

Debido a que el análisis del código total representa el análisis de cientos de líneas de código HTML, se procede a explicar los agregados para los elementos del tipo input y button, ya que, para los restantes elementos, el razonamiento es análogo.

Elemento HTML Input:



- **Código HTML desarrollado en la aplicación web:**

```
<input type="number" class="form-control" id="numeroDos"
name="numeroDos" aria-describedby="numeroDosHelp"
placeholder="Ingrese el Segundo Número" required/>
```

- **Código generado en el navegador web**

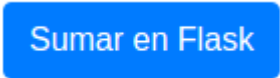
```
<input type="number" class="form-control" id="numeroDos"
name="numeroDos" aria-describedby="numeroDosHelp"
placeholder="Ingrese el Segundo Número" required/>
```

Además del código anterior, en este caso, podemos observar que la clase form-control, del elemento input, implica el agregado de los estilos mostrados a continuación, entre otros cientos de estilos. No tendría sentido realizar un análisis exhaustivo de todos los estilos implicados en la generación del elemento input, sino que lo importante a concluir, es que realmente se agrega información a las vistas html generadas con el framework front end Bootstrap.

- **Información adicional**

```
.form-control {
  display: block;
  width: 100%;
  padding: .5rem .75rem;
  font-size: 1rem;
  line-height: 1.25;
  color: #495057;
  background-color: #fff;
  background-image: none;
  background-clip: padding-box;
  border: 1px solid rgba(0,0,0,.15);
  border-radius: .25rem;
  transition: border-color ease-in-out .15s,box-shadow ease-in-out .15s;
}
```

Elemento HTML Button:



- **Código HTML desarrollado en la aplicación web:**

```
<button type="submit" class="btn btn-primary" style="float: right;">Sumar en Flask</button>
```

- **Código generado en el navegador web**

```
<button type="submit" class="btn btn-primary" style="float: right;">Sumar en Flask</button>
```

Si aplicamos un razonamiento análogo para el elemento button podemos observar como Bootstrap nos simplifica la vida como desarrolladores front end y nos agrega información que implica que el botón luzca más agradable para los usuarios finales de la aplicación. Dicha información agregada está dada por las clases btn y btn-primary.

- **Información adicional**

```
.btn-primary {
  color: #fff;
  background-color: #007bff;
  border-color: #007bff;
}
.btn {
```

```
display: inline-block;
font-weight: 400;
text-align: center;
white-space: nowrap;
vertical-align: middle;
webkit-user-select: none;
moz-user-select: none;
ms-user-select: none;
user-select: none;
border: 1px solid transparent;
padding: .5rem .75rem;
font-size: 1rem;
line-height: 1.25;
border-radius: .25rem;
transition: all .15s ease-in-out;
}
```

Análisis del protocolo HTTP utilizado

En este apartado vamos a analizar que ocurre con el protocolo de aplicación HTTP al momento de utilizar la aplicación web (Sumador de enteros en Flask) desarrollada en este ejercicio. Para ello procedemos a analizar que paquetes HTTP se generan desde el momento en que el usuario inicia la aplicación desde el navegador web y carga los valores a sumar, hasta que, el resultado es visualizado en pantalla.

Para poder analizar dichos paquetes del protocolo de aplicación HTTP, utilizaremos una herramienta llamada Wireshark, disponible para su descarga en su sitio web oficial <https://www.wireshark.org/>.

Antes de continuar, definamos de que se trata la herramienta mencionada, Wireshark, es un analizador de protocolos utilizado para realizar análisis y solucionar problemas en redes de comunicaciones, para desarrollo de software y protocolos, y como una herramienta didáctica. Cuenta con todas las características estándar de un analizador de protocolos de forma únicamente hueca.

La funcionalidad que provee es similar a la de tcpdump, pero añade una interfaz gráfica y muchas opciones de organización y filtrado de información. Así, permite ver todo el tráfico que pasa a través de una red.

Una vez instalada la herramienta en el equipo, ejecutamos la aplicación sumador en Flask. Luego ejecutamos la herramienta Wireshark, seleccionamos que capture los paquetes relacionados con las direcciones de loopback, pues, el sumador se encuentra corriendo en localhost y por ultimo filtramos los paquetes relacionados con el protocolo de aplicación HTTP. A continuación, podemos visualizar este proceso de forma gráfica.



Figura 21

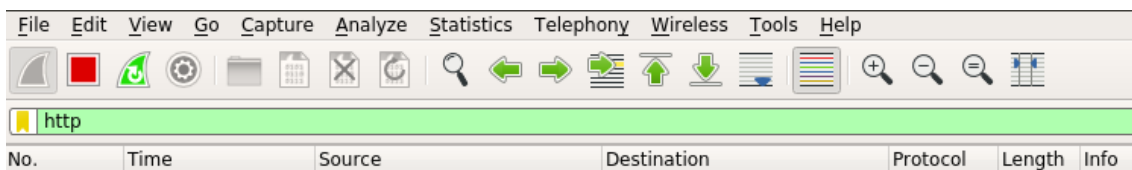


Figura 22

Ahora que tenemos seleccionada la interfaz y protocolo a analizar, procedemos a capturar los paquetes mencionados desde el momento en que el usuario inicia la aplicación web, selecciona los operando a sumar y obtiene el resultado de la suma deseada, es decir siguiendo los pasos de las figuras 18, 19 y 20. Al realizar dicha secuencia de acciones, vemos como se generan los paquetes HTTP desde la herramienta Wireshark.

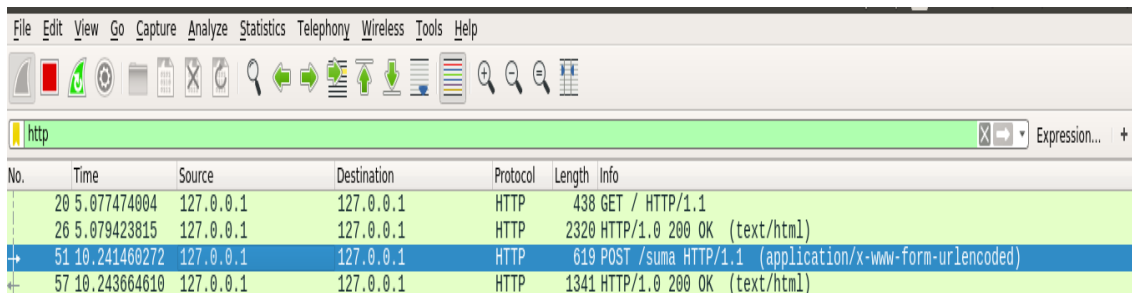


Figura 23

En la figura anterior podemos ver que cuando el usuario inicia la aplicación web, escribiendo en su navegador, la dirección correspondiente, se genera una solicitud HTTP con el verbo GET, la cual es respondida por el servidor Flask, obteniendo un HTTP 200 ok, lo cual significa que se a enviado correctamente la vista al navegador, mostrando el formulario de la suma. Luego de esto, el usuario ingresa los operando a sumar y los envía al servidor de Flask pulsando el botón. Esta acción se corresponde con el tercer paquete filtrado por la herramienta, el cual corresponde a un paquete HTTP con el verbo POST. Este último paquete llega a dicho servidor, generando una respuesta análoga al paquete dos de la lista, pero en este caso, devolviendo el resultado de la suma.

Para concluir podemos analizar el tercer paquete con el fin de visualizar toda la información adicional generada por el servidor Flask, con la cual observaremos la gran cantidad de información generada por el servidor HTTP.

1. En la siguiente imagen vemos la cantidad de bytes capturados y enviados contenidos en el Frame HTTP. Además, vemos la interfaz correspondiente, el tipo de encapsulamiento y el timestamp asociado al paquete.

```
▼ Frame 51: 619 bytes on wire (4952 bits), 619 bytes captured (4952 bits) on interface 0
  Interface id: 0 (lo)
  Encapsulation type: Ethernet (1)
  Arrival Time: Sep 14, 2017 12:41:39.966645696 ART
  [Time shift for this packet: 0.000000000 seconds]
  Epoch Time: 1505403699.966645696 seconds
  [Time delta from previous captured frame: 2.957378394 seconds]
  [Time delta from previous displayed frame: 5.162036457 seconds]
  [Time since reference or first frame: 10.241460272 seconds]
  Frame Number: 51
  Frame Length: 619 bytes (4952 bits)
  Capture Length: 619 bytes (4952 bits)
  [Frame is marked: False]
  [Frame is ignored: False]
  [Protocols in frame: eth:ethertype:ip:tcp:http:urlencoded-form]
  [Coloring Rule Name: HTTP]
  [Coloring Rule String: http || tcp.port == 80 || http2]
```

Figura 24

2. En la imagen siguiente observamos el paquete Ethernet asociado, el cual contiene las direcciones mac del destino y fuente. Cabe mencionar que indica todo ceros debido a que estamos corriendo la aplicación en localhost.

```
▼ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
  ▼ Destination: 00:00:00_00:00:00 (00:00:00:00:00:00)
    Address: 00:00:00_00:00:00 (00:00:00:00:00:00)
    .... 00. .... = LG bit: Globally unique address (factory default)
    .... 00. .... = IG bit: Individual address (unicast)
  ▼ Source: 00:00:00_00:00:00 (00:00:00:00:00:00)
    Address: 00:00:00_00:00:00 (00:00:00:00:00:00)
    .... 00. .... = LG bit: Globally unique address (factory default)
    .... 00. .... = IG bit: Individual address (unicast)
  Type: IPv4 (0x0800)
```

Figura 25

3. A continuación, vemos el header del protocolo de internet IPV4, en donde se muestran los flags correspondientes al mismo junto con las direcciones IP de destino y fuente.

```
▼ Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
  0100 .... = Version: 4
  .... 0101 = Header Length: 20 bytes (5)
  ▼ Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
    0000 00.. = Differentiated Services Codepoint: Default (0)
    .... 0000 = Explicit Congestion Notification: Not ECN-Capable Transport (0)
  Total Length: 605
  Identification: 0x0042 (66)
  ▼ Flags: 0x02 (Don't Fragment)
    0... .... = Reserved bit: Not set
    .1.. .... = Don't fragment: Set
    ..0. .... = More fragments: Not set
  Fragment offset: 0
  Time to live: 64
  Protocol: TCP (6)
  Header checksum: 0x3a57 [validation disabled]
  [Header checksum status: Unverified]
  Source: 127.0.0.1
  Destination: 127.0.0.1
  [Source GeoIP: Unknown]
  [Destination GeoIP: Unknown]
```

Figura 26

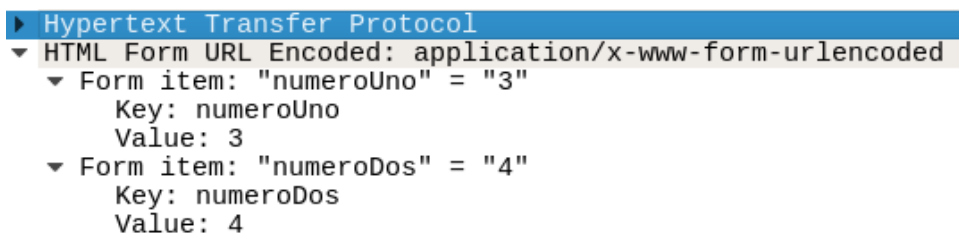
4. En esta nueva figura vemos información del protocolo HTTP como tipo de petición, host y tipo de navegador utilizado.



```
▼ Hypertext Transfer Protocol
  ▶ POST /suma HTTP/1.1\r\n
    Host: 127.0.0.1:8000\r\n
    Connection: keep-alive\r\n
  ▶ Content-Length: 23\r\n
    Cache-Control: max-age=0\r\n
    Origin: http://127.0.0.1:8000\r\n
    Upgrade-Insecure-Requests: 1\r\n
    User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110 Safari/537.36\r\n
    Content-Type: application/x-www-form-urlencoded\r\n
    Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8\r\n
    Referer: http://127.0.0.1:8000/\r\n
    Accept-Encoding: gzip, deflate, br\r\n
    Accept-Language: es-ES,es;q=0.8\r\n
    \r\n
    [Full request URI: http://127.0.0.1:8000/suma]
    [HTTP request 1/1]
    [Response in frame: 57]
    File Data: 23 bytes
```

Figura 27

5. Por último, y por eso no menos importante, podemos ver los valores retornados por la respuesta HPPT POST hacia la vista HTML.



```
▶ Hypertext Transfer Protocol
  ▼ HTML Form URL Encoded: application/x-www-form-urlencoded
    ▼ Form item: "numeroUno" = "3"
      Key: numeroUno
      Value: 3
    ▼ Form item: "numeroDos" = "4"
      Key: numeroDos
      Value: 4
```

Figura 28

Ejercicio 3

Generar un proyecto de simulación de acceso a valores de temperatura, humedad, presión atmosférica y velocidad del viento.

- a) *Un proceso simulará una placa con microcontrolador y sus correspondientes sensores o directamente una estación meteorológica proveyendo los valores almacenados en un archivo o en una base de datos. Los valores se generan periódicamente (frecuencia de muestreo).*
- b) *Un proceso generará un documento HTML conteniendo:*
 - i. *Frecuencia de muestreo*
 - ii. *Promedio de las últimas 10 muestras*
 - iii. *La última muestra*
- c) *El documento HTML generado debe ser accesible y responsivo.*

Aclaración: Se deberá detallar todo el proceso de adquisición de datos, cómo se ejecutan ambos procesos (ya sea threads o procesos separados), el esquema general, las decisiones tomadas en el desarrollo de cada proceso y la interacción del usuario.

Interpretación general

El objetivo de este ejercicio es realizar una simulación de acceso a un conjunto determinado de variables meteorológicas. Se debe simular el proceso completo, desde la generación de los datos, el almacenamiento de los mismos, hasta la presentación de un documento accesible para el usuario final.

Las variables meteorológicas que se tendrán en cuenta serán las siguientes

- Temperatura [°C]
- Humedad [%]
- Presión atmosférica [hPa]
- Velocidad del viento [km/h]

Para el proyecto se utilizará un modelo *cliente-servidor*:

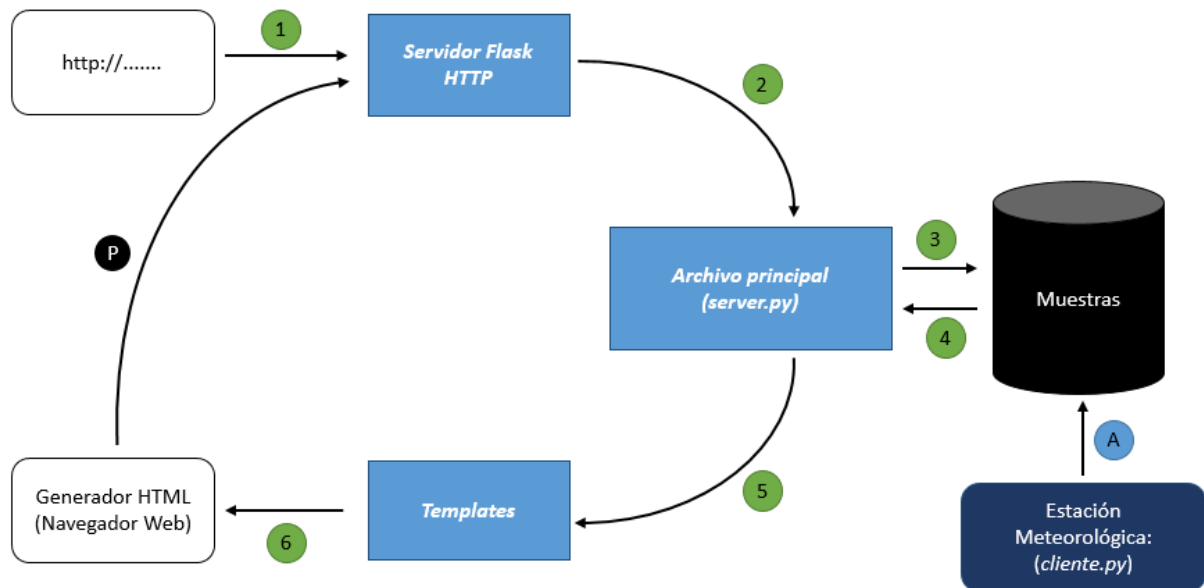
- Cliente: es un proceso que simula una placa con microcontrolador y sus correspondientes sensores. Las lecturas realizadas por los sensores se simulan generando números aleatorios (en rangos razonables según la variable que se esté midiendo) en un período fijo de 5 segundos
- Servidor: presenta la información a través de un documento HTML, accesible para el usuario y con un formato que facilite la interpretación de los datos.

Para almacenar los datos, se utilizará sistema de gestión de base de datos relacional, en particular, MySQL. A través del mismo, se creará una base de datos, con una tabla que contenga las mediciones.

El cliente, generará los datos mediante un programa escrito en el lenguaje de programación Python, y la información se presentará a través de un documento HTML en un servidor con Flask. A continuación, se profundizará en la solución desarrollada.

Esquema de funcionamiento

El siguiente esquema muestra, resumidamente, como es la interacción general de todos los componentes del sistema.



La información se encuentra centralizada en la base de datos, donde se crea una tabla que contiene toda la información de las muestras.

La estación meteorológica (que corresponde al archivo *cliente.py*), periódicamente envía nueva información que es allí almacenada (cada 5 segundos, ocurre la interacción indicada en **A**).

La sección izquierda del diagrama, muestra todo el proceso desde que el usuario intenta acceder a la información, hasta que esta es presentada en su pantalla. Como se indica en la figura, este proceso puede desagregarse en 6 etapas:

1. El usuario hace una *request* desde su navegador a una URL en particular, que es atendido por el servidor Flask. En este ambiente de simulación, se correrá localmente el servidor en un puerto de la computadora sobre la que se simula (en particular, puerto 7000: *localhost:7000*)
2. El servidor Flask, atiende la petición realizada, y se encarga de correr el archivo principal *server.py*
3. El archivo en cuestión, realiza una consulta a la base de datos para obtener la información generada por la estación meteorológica.
4. El sistema de gestión de base de datos, procesa la consulta realizada y devuelve la información al programa
5. Con la información obtenida, se renderiza un template HTML que presenta un formato bien definido, accesible para el usuario.
6. Finalmente, el archivo HTML es tomado por el navegador web, nuevamente del lado del usuario, para presentarle los datos.

Notar que la estación meteorológica genera información periódicamente, por lo que transcurrido un determinado periodo la información que está observando el usuario estará desactualizada. En este caso, esto sucederá cada 5 segundos.

Para prevenir esto, se utilizó la biblioteca *jQuery*, que se encarga de realizar automáticamente consultas periódicas al servidor para mantener actualizada la página (indicado en el diagrama como **P**). De este modo, el usuario puede contar con información actualizada sin necesidad de refrescar su página.

Los scripts se ejecutan periódicamente, cada 5 segundos. Al ejecutarse, realizan un *request* a una ruta específica del servidor flask, que genera la ejecución de las sentencias SQL. Con la información, se modifica únicamente la sección del documento HTML asociada a estos datos.

```
<script type="text/javascript">
    function actualizaUltima(){
        $('#ultima').load('ultima');
        setTimeout( actualizaUltima , $frec*1000 )
    }
    $(document).ready( actualizaUltima );
</script>
```

El script anterior, por ejemplo, muestra como se realiza la actualización de la última muestra. Se ejecuta la función “actualizaUltima”, que cada 5000ms (\$frec por defecto vale 5seg, y se pasa a milisegundos), carga una porción de HTML contenida en localhost:7000/ultima.

En *server.py* se crea la ruta ultima, que ejecuta la consulta al servidor y devuelve la porción de HTML con los datos necesarios:

```
@app.route('/ultima')
def ultima():
    conn = mysql.connect()
    cursor = conn.cursor()
    cursor.execute('SELECT * from Medidas ORDER BY idMedida desc LIMIT 1')
    lastData = cursor.fetchone()

    string = '<th scope="row">' + str( round( lastData[0] , 2 ) ) + '</th>'
    string = string + '<td>' + str( round( lastData[1] , 2 ) ) + ' °C</td>'
    string = string + '<td>' + str( round( lastData[2] , 2 ) ) + ' % </td>'
    string = string + '<td>' + str( round( lastData[3] , 2 ) ) + ' hPa </td>'
    string = string + '<td>' + str( round( lastData[4] , 2 ) ) + ' km/h </td>'

    return string
```

De la misma forma, se crean funciones en javascript que mantienen actualizado el promedio de las últimas 10 muestras, y el historial; sin necesidad de que el usuario actualice manualmente el sitio.

La interfaz presentada al usuario, será como se muestra a continuación:

Taller de Proyecto II

Ejercicio 3

Periodo Actual: 5 [Segundos]

Promedios:

#	Temperatura	Humedad Relativa	Presión Atmosférica	Viento
Últimas 10 muestras:	6.65 °C	49.14 %	896.18 hPa	208.67 km/h

Ultima Muestra:

Medida ID	Temperatura	Humedad Relativa	Presión Atmosférica	Viento
179	72.85 °C	15.99 %	947.35 hPa	382.7 km/h

Historico de Medidas:

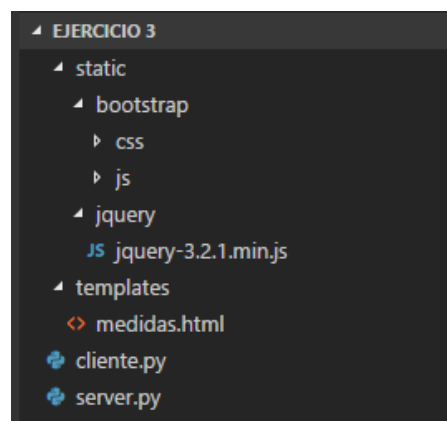
Medida ID	Temperatura	Humedad Relativa	Presión Atmosférica	Viento
1	33.0 °C	45.0 %	1023.0 hPa	12.0 km/h
2	38.0 °C	48.0 %	1022.0 hPa	10.0 km/h
3	39.0 °C	42.0 %	1020.0 hPa	18.0 km/h
4	36.0 °C	52.0 %	1025.0 hPa	6.0 km/h
5	41.39 °C	29.69 %	859.43 hPa	209.4 km/h
6	-11.04 °C	14.45 %	1169.84 hPa	335.33 km/h

Herramientas necesarias

Para poder desarrollar el siguiente proyecto fue necesario:

1. Flask: para levantar el servidor, y librería para conectarse con la base de datos
2. Python: para correr los programas en dicho lenguaje
3. Bootstrap y JQuery: para mejorar accesibilidad del documento HTML
4. MySQL: para diseñar y correr la base de datos

El programa se diseñó con la siguiente estructura de carpetas:



Se puede observar que, dentro de la carpeta *static*, se incluyen las subcarpetas *bootstrap* y *jquery*. En ellas, se ubican todos los archivos necesarios para poder utilizar las librerías en cuestión desde el archivo HTML: *medidas.html*. Estos archivos, se descargan de las respectivas páginas oficiales:

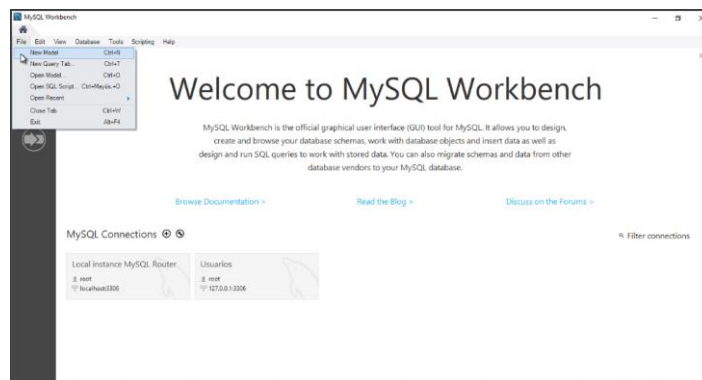
- <http://getbootstrap.com/>
- <https://jquery.com/>

Por otro lado, para diseñar y correr la base de datos, se empleó MySQL. El primer paso para ello, consistió en descargarlo e instalarlo siguiendo las configuraciones recomendadas:

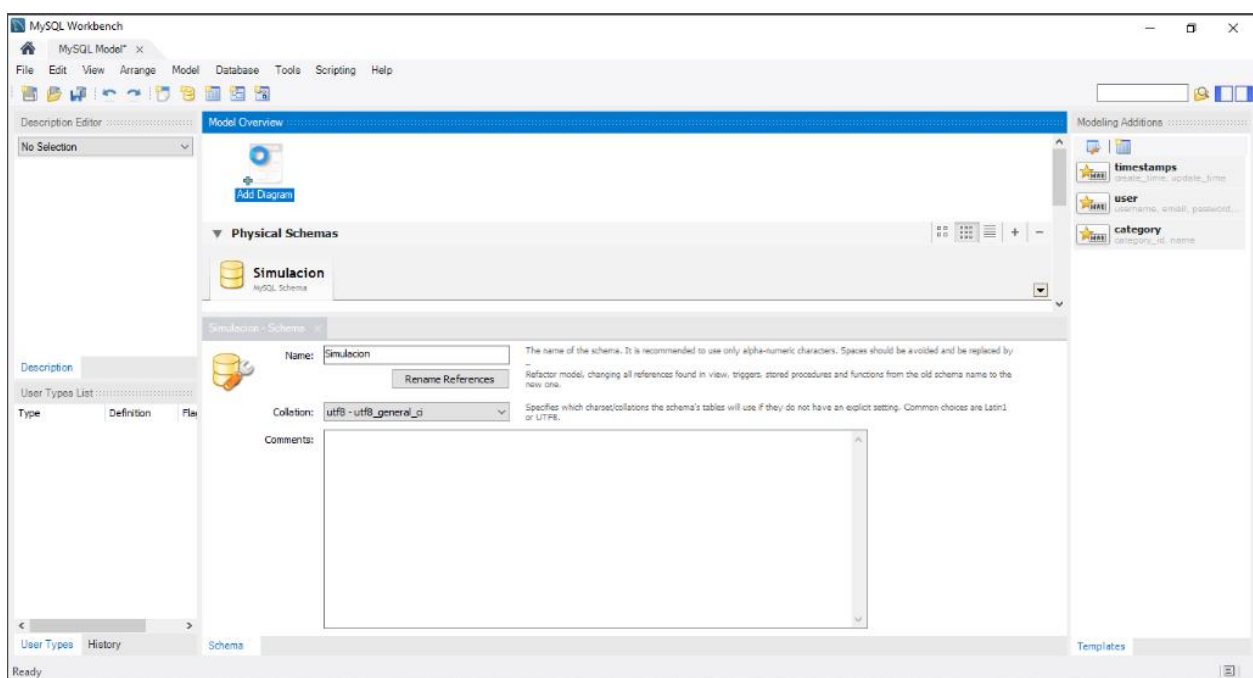
- <https://www.mysql.com/>

Una vez instalado, para diseñar la base de datos se siguieron los siguientes pasos:

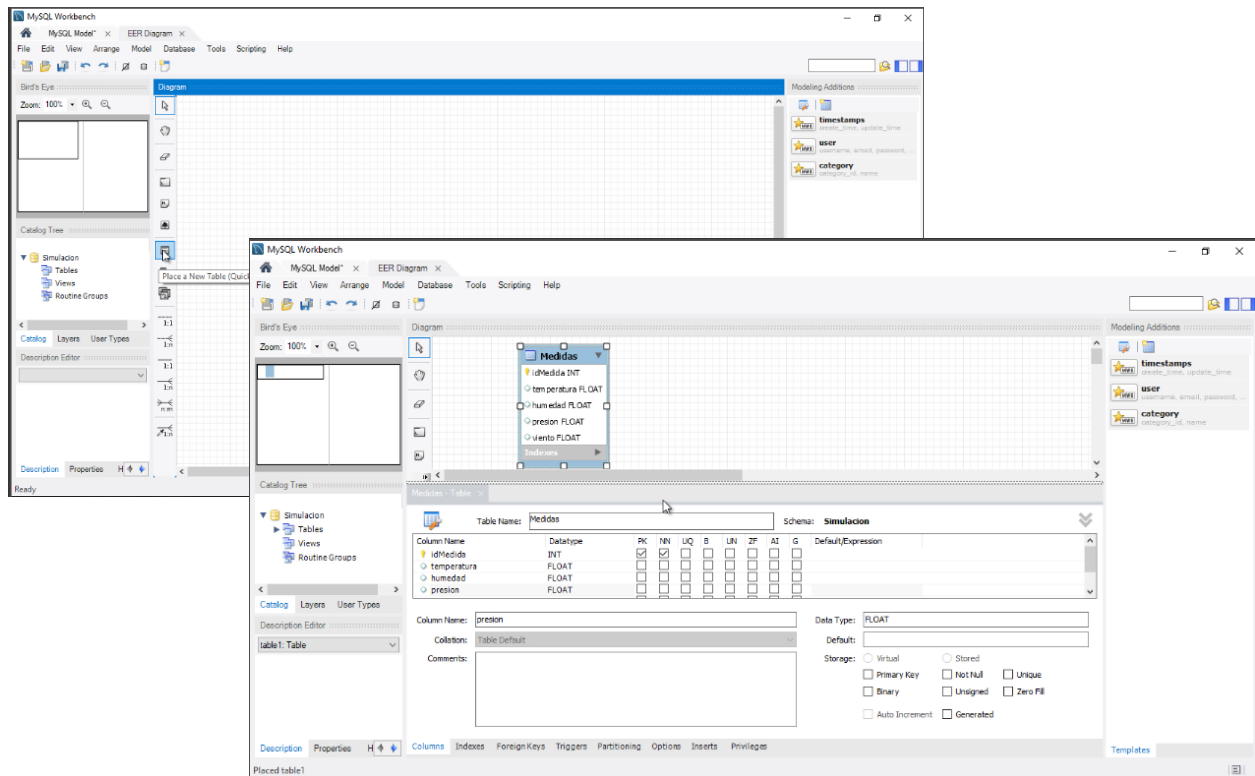
1. Abrir MySQL workbench y seleccionar nuevo modelo



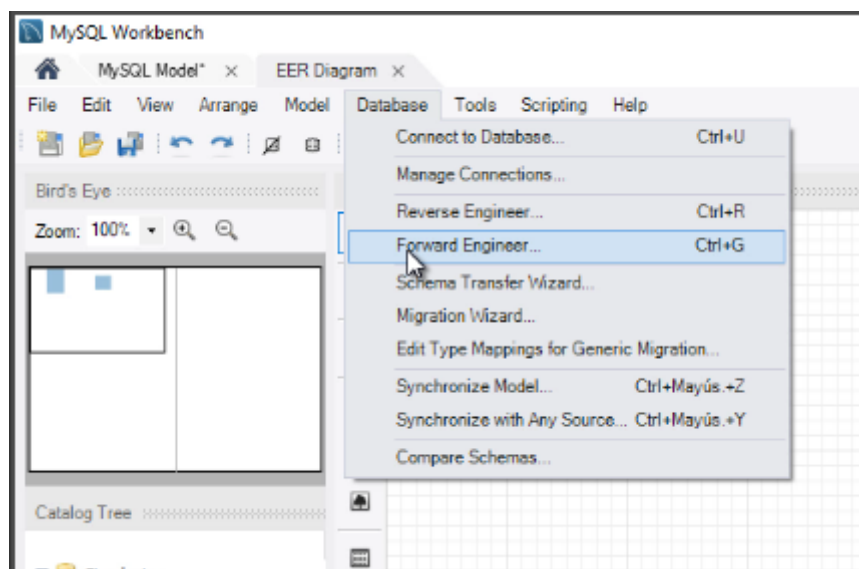
2. Seleccionar el nombre para la base de datos, en este caso “Simulación”, y crear un nuevo diagrama



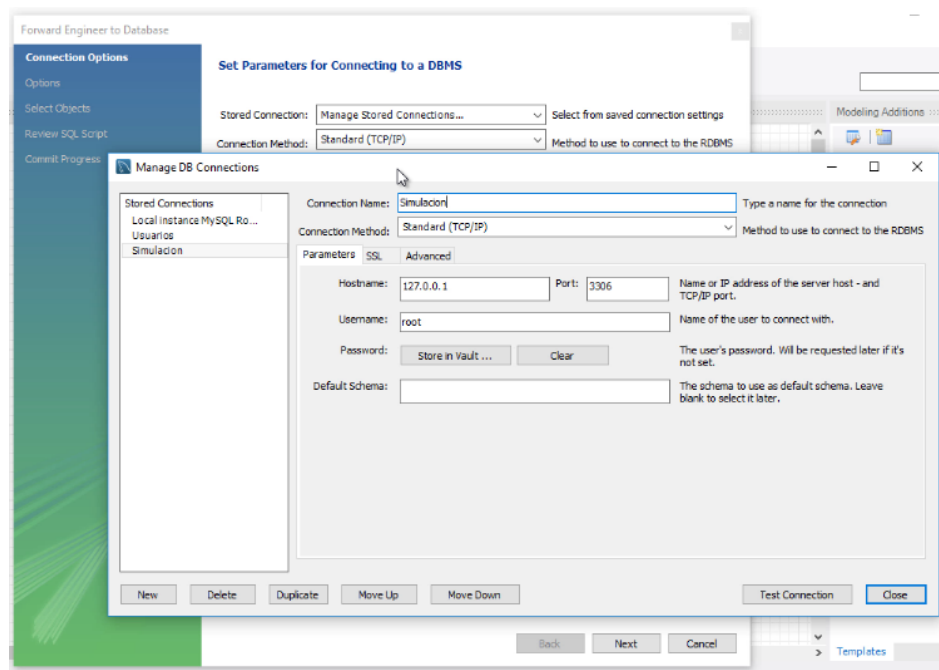
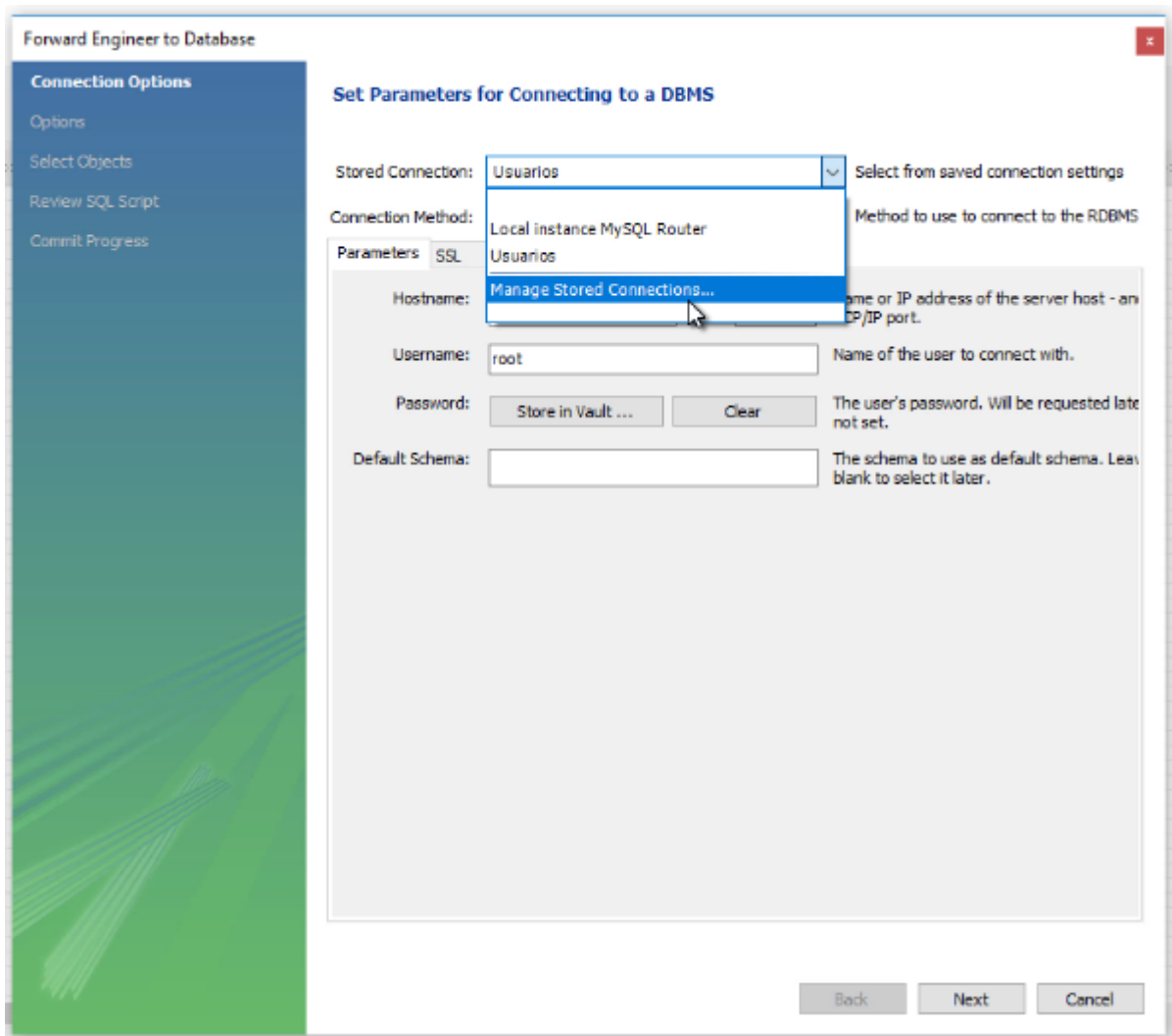
3. Dentro del diagrama, es posible crear las tablas que sean necesarias. En este caso, se crea una única tabla que almacena las muestras, como se muestra a continuación:



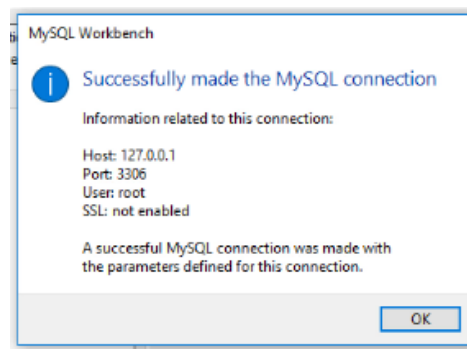
4. Una vez que el diseño ha sido completado, se procede a crear efectivamente las tablas necesarias. Para ello, se selecciona "Forward Engineer..."



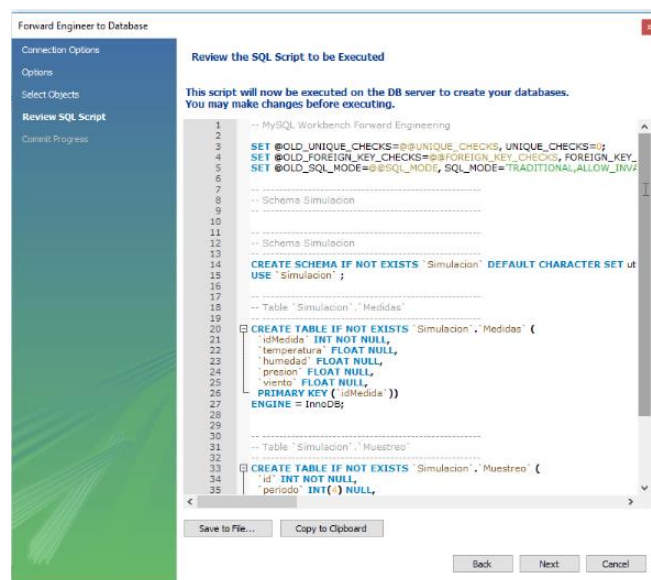
5. Se abrirá una nueva ventana con los parámetros para crear la base. Primeramente, se seleccionará la conexión. En esta ocasión se crea una nueva, como muestran las siguientes dos figuras.



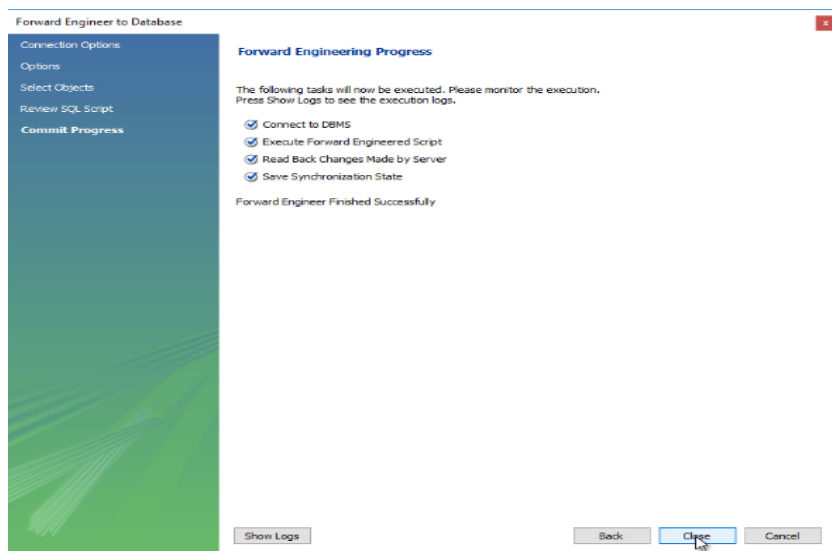
6. Para confirmar que la conexión se creó exitosamente, se verifica a través de “Test Connection”. El siguiente cuadro de diálogo, confirma que en *localhost* (127.0.0.1), en el puerto 3306 el SGBD podrá atender solicitudes para esta base de datos



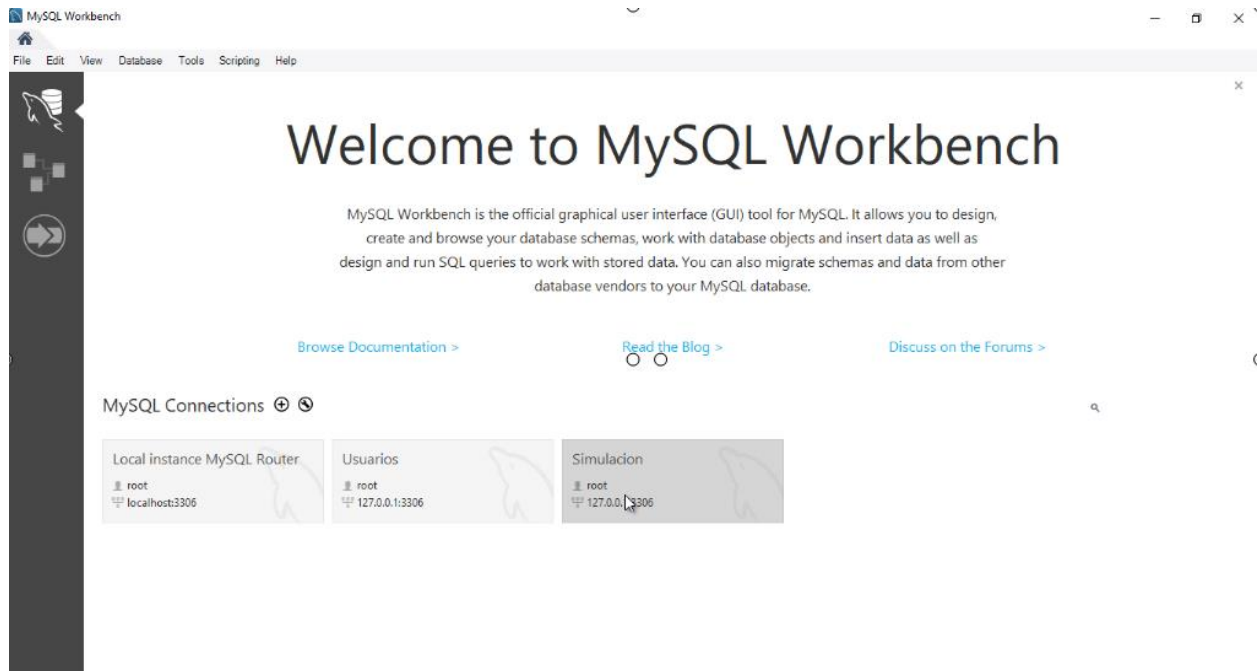
7. El proceso de Forward Engineer, una vez creada la conexión, muestra las instrucciones SQL que se ejecutarán para crear la base



8. Finalmente, el asistente crea la base, conectándose con el SGBD y ejecutando las instrucciones en cuestión



9. Al abrir el Workbench, se puede observar que se encuentra activa la conexión que se ha creado para acceder a la base con las muestras



Replicar funcionamiento

Junto con el presente informe, se adjuntan todos los archivos necesarios para replicar la simulación. Los pasos a seguir son los siguientes:

1. Instalar MySQL, y crear la base de datos tal como se detallo anteriormente.
2. Dentro de la carpeta “Ejercicio 3”:
 - a. Ejecutar el archivo *server.py* que se encargará de levantar el servidor Flask
 - b. Ejecutar el archivo *client.py* que se encargará de generar las muestras.
3. Abrir algún navegador web, y acceder a: *localhost:7000*

Una vez hecho esto, se podrá observar la información en cuestión, así como su proceso de actualización automática cada 5 segundos.

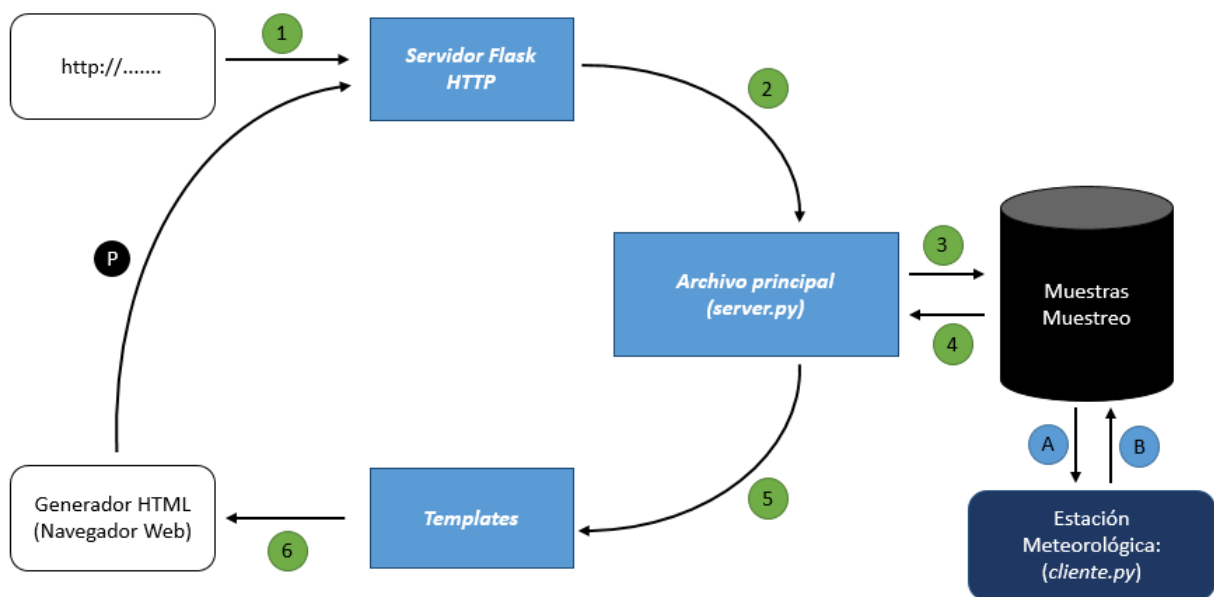
Ejercicio 4

Agregar a la simulación anterior la posibilidad de que el usuario elija entre un conjunto predefinido de períodos de muestreo (ej.: 1, 2, 5, 10, 30, 60 segundos). Identifique los cambios a nivel de HTML, de HTTP y de la simulación misma.

Resolución

Para la resolución de este ejercicio, se tomará el proyecto desarrollado en el ejercicio anterior, agregando la opción de cambiar la frecuencia de muestreo.

El siguiente esquema, resume la nueva implementación realizada:



Como se puede observar, es sumamente similar a la utilizada para el Ejercicio 3. La diferencia principal, radica en la inclusión de una nueva tabla en la base de datos, que contiene el periodo de muestreo con el que se trabaja.

Para permitir que la estación meteorológica pueda ajustarse al periodo de muestreo seleccionado por el usuario, se agrega esta tabla en la base de datos que es accesible tanto por el servidor Flask como por la estación meteorológica.

De esta forma, el programa *cliente.py* ya no solo escribe en la base de datos, sino que también lee el periodo actual para saber cada cuánto tiempo debe generar una nueva muestra.

A su vez, el archivo HTML debe ser modificado para que el usuario pueda seleccionar diferentes periodos de muestreo, y que su elección se cargue en la base de datos.

Nuevamente, esto se resuelve utilizando JQuery: se agrega un menú de opción múltiple con los periodos de muestreo posibles, y un botón para establecerla. Cuando el usuario presiona el botón, el script de JQuery captura el evento, y hace un *request* a una ruta específica del servidor flask, que ejecuta la sentencia SQL para actualizar la base de datos.


```

<script type="text/javascript">
    $(document).ready(function() {
        $("button").click(function() {
            $frec = $("#exampleFormControlSelect1").val();
            $("#frecMuestreo").load('periodo/' + $frec);
            console.log("Periodo de muestreo: " + $frec );
            actualizaPromedios();
            actualizaUltima();
            actualizaHistorico();
        });
    });
</script>

```

El script anterior se encarga de realizar esta tarea: asociado al elemento “Button”, cuando se hace click en el mismo, se hace una consulta a la ruta localhost:7000/periodo/\$frec, con \$frec conteniendo la opción seleccionada por el usuario.

```

@app.route('/periodo/<int:p>')
def periodo(p):
    if( p in [1,2,5,10,30,60] ):
        muestreo = p
        conn = mysql.connect()
        cursor = conn.cursor()
        cursor.execute('UPDATE `Muestreo` SET `periodo`=' + str(p) + ' WHERE 1')
        conn.commit()

    if( muestreo == 1 ):
        return "<h4>Periodo Actual: " + str(muestreo) + " Segundo</h4>"
    else:
        return "<h4>Periodo Actual: " + str(muestreo) + " Segundos</h4>"

```

Dentro de *server.py*, el código anterior se encarga de cargar en la base de datos la nueva frecuencia de muestreo, y devolver la porción HTML indicando el nuevo periodo seleccionado.

La página con la opción de selección de periodo de muestreo presenta la siguiente interfaz:

Taller de Proyecto II

Ejercicios 3 y 4

Periodo Actual: 30 [Segundos]

Seleccionar Periodo de Muestreo:

1

Nuevo Periodo

Promedios:

#	Temperatura	Humedad Relativa	Presión Atmosférica	Viento
Últimas 10 muestras:	6.65 °C	49.14 %	896.18 hPa	208.67 km/h

Ejercicio 5

Comente la problemática de la concurrencia de la simulación y específicamente al agregar la posibilidad de cambiar el periodo de muestreo. Comente lo que estima que podría suceder en el ambiente real. ¿Podrían producirse problemas de concurrencia muy difíciles o imposibles de simular? Comente brevemente los posibles problemas de tiempo real que podrían producirse en general.

Resolución

En primer lugar, la simulación plantea un problema de concurrencia desde el comienzo, dado que las muestras deben ser accedidas al mismo tiempo que se están creando. Esto ocasiona que un mismo archivo sea accedido al mismo tiempo, para lectura y escritura de datos.

La solución implementada para esta situación, consistió en emplear un Sistema de Gestión de Base de Datos (MySQL). De esta forma, entre las tareas propias de un SGBD se encuentra el control de la concurrencia a la base de datos. Así, podemos garantizar que ninguna lectura o escritura pueda llegar a alterar la consistencia de la información.

Por otro lado, al agregar la posibilidad de cambiar la frecuencia de muestreo, se genera un nuevo problema de concurrencia: al mismo momento que el usuario selecciona un periodo determinado, el generado de muestras debe conocerlo, o caso contrario, se generaría una discrepancia entre el periodo seleccionado por el usuario y la generación de las muestras.

Nuevamente, la solución adoptada consistió en crear una tabla en la base de datos, que permita delegar el control de la concurrencia al SGBD. Así, tanto el cliente (generador de muestras), como el servidor pueden compartir la información sobre el periodo de muestreo sin generar inconsistencias.

Como aclaración de lo anterior, vale remarcar que en ciertos escenarios se generarán de todas formas algunos desfases transitorios. Si el usuario reduce el período de muestreo (por ej. de 60 a 10 segundos), puede que el cliente se encuentre en modo *sleep* esperando el *timeout* del periodo actual, y recién cuando este se produzca podrá verificar que el período de muestreo ha cambiado.

Este escenario, solo sucederá cuando el periodo de muestreo disminuye, y tan solo ocurrirá para la primera muestra. Si resulta crucial que esto no suceda, puede configurarse el cliente para que verifique continuamente el periodo de muestreo, generando una mayor cantidad de consultas a la base de datos, pero garantizando una respuesta más veloz.

En un ambiente real, los problemas de concurrencia mencionados también tendrían lugar. A su vez, es razonable considerar que el cliente se ubicaría físicamente alejado del servidor, por lo que habría un retardo mayor debido a la mera latencia de comunicación.

Sumado a esto, pueden existir problemas externos al sistema difíciles de solucionar, tales como la falta de señal entre el cliente y el servidor. Si la estación meteorológica se encuentra ubicada en zonas no urbanas, puede que la escasa cobertura haga imposible la conexión en tiempo real a un servidor remoto.

Ejercicio 6

¿Qué diferencias supone que habrá entre la simulación planteada y el sistema real? Es importante para planificar un conjunto de experimentos que sean significativos a la hora de incluir los elementos reales del sistema completo.

Resolución

En primer lugar, en el sistema real deberán considerarse las especificaciones de los sensores a utilizar, que en esta simulación se limitaron a contemplarse como variables aleatorias.

Por otro lado, como se mencionaba en el punto anterior, es razonable pensar que el cliente y el servidor se encuentren físicamente separados, por lo que debería contemplarse la comunicación. De la misma forma, la base de datos podría situarse en otro lugar, diferente al cliente y el servidor.

Se piensa en la escalabilidad del sistema, se pueden tener múltiples clientes, ubicados en diferentes zonas, lo cual implicaría un mayor congestionamiento de la base de datos para manejar todas las peticiones en simultáneo.

Para poder tener una mejor aproximación acerca del funcionamiento del sistema en un contexto real, sería conveniente desarrollar un experimento en el cual se utilice un sensor real, conectado a algún microcontrolador o Arduino. Estos valores, deberían ser transmitidos mediante algún protocolo, y almacenados en la base de datos.

De esta forma, podrían notarse los problemas y complejidades asociadas a la transmisión de los datos, ruido de la comunicación, consideraciones energéticas (alimentación del microcontrolador). Utilizando varias placas, es posible expandir el experimento a un escenario con múltiples clientes, enfrentando un escenario más próximo al real.