

Informe Práctica 1

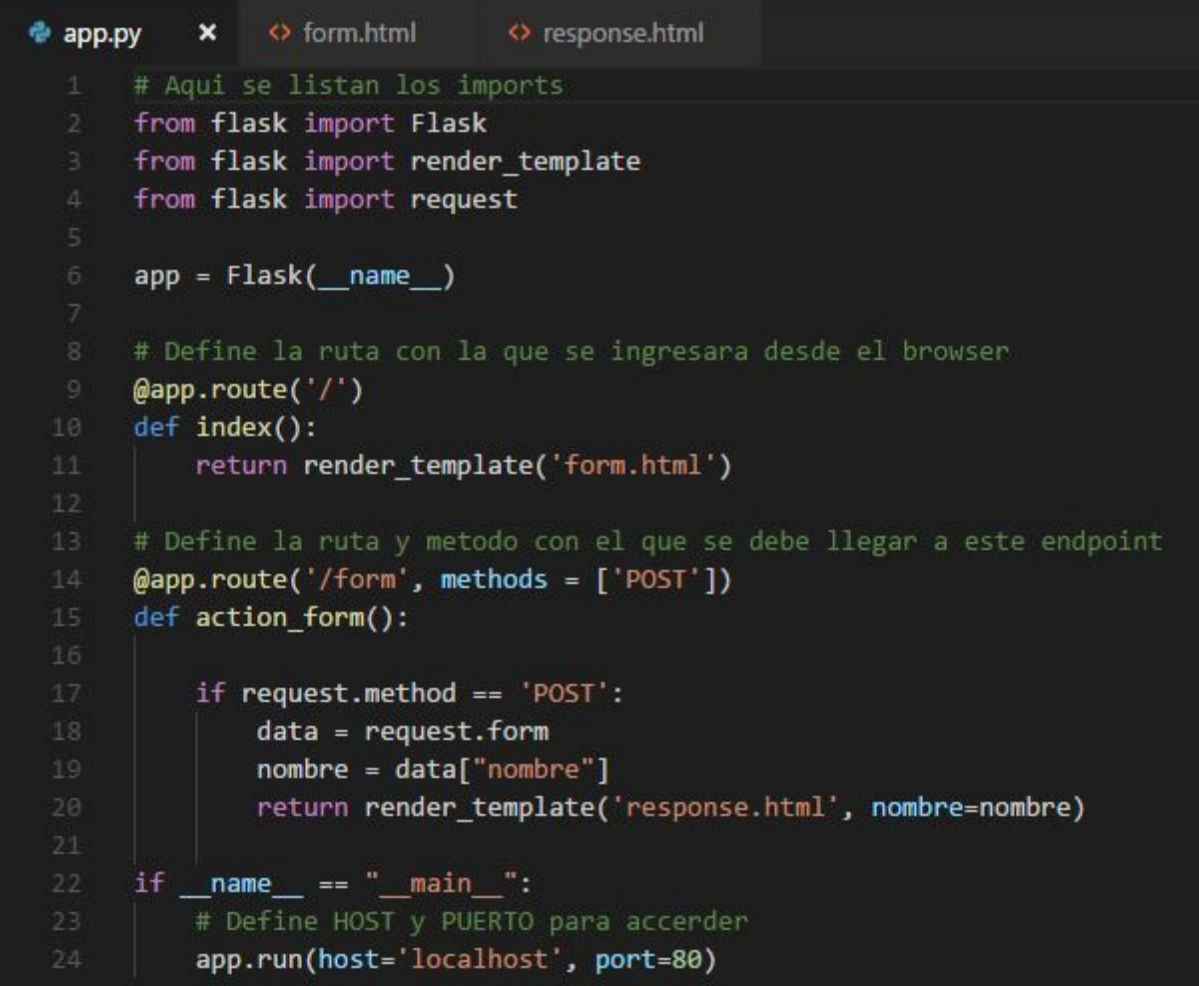
Taller de Proyecto II

Integrantes:

- Jorge Luis Stranieri, N°: 917/5.
- Matias Iglesias, N°: 828/5

- 1) Las dependencias necesarias para poder levantar un servidor con Flask son dos:
 - a) Instalar Python 2.7 o superior
 - b) Una vez que tenemos Python instalado hay que instalar pip, esta es una herramienta que nos permite hacer la instalación de forma sencilla de los paquetes para python y desde la consola. Con pip luego instalamos Flask.

Ejemplo:



```
1 # Aqui se listan los imports
2 from flask import Flask
3 from flask import render_template
4 from flask import request
5
6 app = Flask(__name__)
7
8 # Define la ruta con la que se ingresara desde el browser
9 @app.route('/')
10 def index():
11     return render_template('form.html')
12
13 # Define la ruta y metodo con el que se debe llegar a este endpoint
14 @app.route('/form', methods = ['POST'])
15 def action_form():
16
17     if request.method == 'POST':
18         data = request.form
19         nombre = data["nombre"]
20         return render_template('response.html', nombre=nombre)
21
22 if __name__ == "__main__":
23     # Define HOST y PUERTO para acceder
24     app.run(host='localhost', port=80)
```

Este es un ejemplo que se explico en la práctica, en la cual se utiliza Flask para levantar un servidor que permita realizar las acciones que se definen en este archivo python. Acá se define lo principal que da inicio a la interacción usuario-servidor, que es el host=localhost ó 0.0.0.0 y el puerto, en este caso port=80. A grandes rasgos éstas son las secuencias de acciones que se llevan a cabo desde el inicio de la interacción del usuario:

- El usuario ingresa a la página, en este caso localhost, ésta es la primer acción que dispara las siguientes, justamente el usuario está solicitando ingresar en esta página.
- Una vez aceptada la solicitud del usuario al querer ingresar a la página, el servidor muestra por defecto el contenido de *form.html* como está dispuesto en el código.
- Hasta este punto no hay nada más ejecutándose debido a que en este ejemplo, la vista que se muestra es la de un formulario para ingresar datos, a menos que el usuario no interactúe directamente no se ejecuta ningún proceso en background o acción, pero podría ser que esto ocurra.

- Una vez que el usuario ingresa información al formulario e interactúa con él, se manda otra solicitud al servidor que es captada por la función *action_form()* que es la que define la respuesta del servidor a esta petición, que ésta vez la respuesta del servidor es mostrar los datos ingresados por el usuario mediante el formulario en una vista simple.

Esta sería la secuencia de acciones o “procesos” que se llevarían a cabo.

2) Siguiendo con el ejemplo anterior, a el usuario no se le muestra qué tipo de método HTTP se le está enviando al servidor si ingresa información mediante el formulario, o el id de la información enviada por el usuario. Para mostrar la información que agrega el protocolo HTTP que genera un programa hecho en Python, utilizamos dos programas para mostrar esto adecuadamente. Los programas son: *WireShark* y *Rawcap*.

WireShark es un programa para analizar los protocolos de cualquier conexión o red. RawCap es un sniffer que al ser ejecutado captura los paquetes que se estén transmitiendo en la red que le indiquemos. En nuestro caso lo usamos para capturar los paquetes que se transmiten desde que levantamos nuestro servidor en localhost. Luego de que se capturen los paquetes, RawCap guarda esta información en una instantánea que luego abrimos con WireShark para ver su contenido.

Para poder ver esto, lo primero que tenemos que hacer es levantar el servidor y entrar en localhost. Una vez hecho esto, iniciamos RawCap desde la línea de comandos:

- `RawCap.exe 127.0.0.1 localhost_capture.pcap`

La dirección 127.0.0.1 le indica a RawCap que tiene que estar escuchando la interfaz de Área Local, ya que es donde levantamos nuestro servidor, y “*localhost_capture.pcap*” es el nombre de la instantánea en la cual se guardarán los datos que capture Rawcap que serán de los paquetes transmitidos. Luego habiendo tenido instalado WireShark abrimos la instantánea para ver la información. Esto muestra la información que agrega el protocolo HTTP a un programa hecho en python:

```
Wireshark - Packet 54 - localhostcapture

> Frame 54: 526 bytes on wire (4208 bits), 526 bytes captured (4208 bits)
Raw packet data
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
> Transmission Control Protocol, Src Port: 63620, Dst Port: 8765, Seq: 1, Ack: 1, Len: 486
▼ Hypertext Transfer Protocol
  > GET / HTTP/1.1\r\n
    Host: localhost:8765\r\n
    Connection: keep-alive\r\n
    Cache-Control: max-age=0\r\n
    Upgrade-Insecure-Requests: 1\r\n
    User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/61.0.3163.100 Safari/537.36\r\n
    Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8\r\n
    Referer: http://localhost:8765/\r\n
    Accept-Encoding: gzip, deflate, br\r\n
    Accept-Language: en-US,en;q=0.8,es-419;q=0.6,es;q=0.4,tr;q=0.2,gl;q=0.2\r\n
    \r\n
    [Full request URI: http://localhost:8765/]
    [HTTP request 1/1]
    [Response in frame: 70]

0000 45 00 02 0e 0d 10 40 00 80 06 00 00 7f 00 00 01 E....@. ....
0010 7f 00 00 01 f8 84 22 3d f1 40 2e 35 a8 8e 02 ae .....= .@.5...
0020 50 18 08 05 97 1c 00 00 47 45 54 20 2f 20 48 54 P..... GET / HT
0030 54 50 2f 31 2e 31 0d 0a 48 6f 73 74 3a 20 6c 6f TP/1.1.. Host: lo
0040 63 61 6c 68 6f 73 74 3a 38 37 36 35 0d 0a 43 6f calhost: 8765..Co
0050 6e 6e 65 63 74 69 6f 6e 3a 20 6b 65 65 70 2d 61 nnection : keep-a
0060 6c 69 76 65 0d 0a 43 61 63 68 65 2d 43 6f 6e 74 live..Ca che-Cont
0070 72 6f 6c 3a 20 6d 61 78 2d 61 67 65 3d 30 0d 0a rol: max -age=0..
0080 55 70 67 72 61 64 65 2d 49 6e 73 65 63 75 72 65 Upgrade- Insecure
0090 2d 52 65 71 75 65 73 74 73 3a 20 31 0d 0a 55 73 -Request s: 1..Us
00a0 65 72 2d 41 67 65 6e 74 3a 20 4d 6f 7a 69 6c 6c er-Agent : Mozill
00b0 61 2f 35 2e 30 20 28 57 69 6e 64 6f 77 73 20 4e a/5.0 (W indows N
00c0 54 20 31 30 2e 30 3b 20 57 69 6e 36 34 3b 20 78 T 10.0; Win64; x
00d0 36 34 29 20 41 70 70 6c 65 57 65 62 4b 69 74 2f 64) Appl eWebKit/
00e0 35 33 37 2e 33 36 20 28 4b 48 54 4d 4c 2c 20 6c 537.36 ( KHTML, l
00f0 69 6b 65 20 47 65 63 6b 6f 29 20 43 68 72 6f 6d ike Geck o) Chrom
0100 65 2f 36 31 2e 30 2e 33 31 36 33 2e 31 30 30 20 e/61.0.3 163.100
0110 53 61 66 61 72 69 2f 35 33 37 2e 33 36 0d 0a 41 Safari/5 37.36..A
0120 63 63 65 70 74 3a 20 74 65 78 74 2f 68 74 6d 6c ccept: t ext/html

No.: 54 · Time: 10.593954 · Source: 127.0.0.1 · Destination: 127.0.0.1 · Protocol: HTTP · Length: 526 · Info: GET / HTTP/1.1
```

Para mostrar esta información que agrega el protocolo HTTP la mostramos en base al ejercicio hecho para la práctica, pero se puede ver de cualquier programa hecho en python, el ejemplo más sencillo sería un programa que imprime "Hola mundo!" simplemente. Lo que hay que hacer para capturar estos datos de forma adecuada, usando *Rawcap* y *WireShark* es:

- Primero nos aseguramos de tener instalados los dos programas, su descarga es gratuita y su instalación es muy sencilla. El programa RawCap es portable, no hay que hacer ninguna instalación, solamente saber el directorio en el cual está instalado y ejecutar desde allí el comando mostrado anteriormente.
- Una vez instalados estos programas, levantamos el servidor con cualquier proyecto hecho, entramos desde el navegador para ver la información y abrimos la línea de comando y ejecutamos el comando de RawCap antes mencionado. Después de esto se van a ir mostrando los paquetes capturados, al mismo tiempo se puede interactuar con la página, así otros métodos pueden ser captados también.
- Lo siguiente es terminar con la ejecución del servidor apretando "CTRL+C" de la terminal en la cual lo iniciamos, y lo mismo para la terminal en donde iniciamos la ejecución de RawCap. Al finalizar esto, RawCap guarda todos los paquetes en una instantánea en el mismo directorio en donde se encuentra, lo único que queda es abrir la instantánea que para esto instalamos WireShark en este caso.

3) El esquema general para la resolución del ejercicio que proponemos, tiene en cuenta dos procesos que se ejecutan al mismo tiempo. La generación en forma aleatoria y guardado (en el archivo de texto llamado "datos.txt") de los datos meteorológicos que se quiere mostrar por un lado, y por otro lado la obtención de los mismos para ser mostrados

a través de la página web. Para esto se debe ejecutar en una consola el archivo llamado "main.py", el cual será el proceso que simula los sensores; y luego se debe ejecutar a través de otra consola el archivo "web.py", el cual se encargara de montar el sitio web al que se podrá acceder a través de un navegador ingresando la dirección "0.0.0.0:8765".

El proceso de generación de datos es simplemente un bucle que ejecuta funciones que generan valores random simulando ser los datos meteorológicos (estas funciones son incluidas al comienzo del archivo, provistas por la librería "random"), y se guardan en un archivo de texto, simulando una "base de datos". Al iniciar este proceso (cada vez que se ejecute) se realizará la apertura del archivo que contiene los datos de forma tal que en caso de que este archivo contenga alguna información guardada, esta sea eliminada (todo el manejo del archivo "datos.txt" se realiza utilizando las funciones que provee el lenguaje Python de forma nativa). Cada muestra (la cual contiene los valores de los cuatro sensores) generada se guarda en el archivo de texto en una misma línea, separando los valores de cada sensor por guiones (ejemplo: 30-15-40-90); luego se efectúa un salto de línea. Si bien cada valor generado podría pertenecer a cualquiera de los cuatro parámetros, tomamos como convención para la escritura/lectura sobre el archivo el siguiente orden: presión-temperatura-humedad-velocidadDelViento. Además, para simplificar su uso se definió la clase Muestra (contenida en "muestra.py"), la cual simplemente se compone por 4 atributos que representan a cada uno de los parámetros climáticos; y además un método "__add__" que se encarga de que al realizar la suma (con el operando "+") de dos variables de tipo Muestra, se devuelva como resultado de la operación una variable de tipo Muestra con la suma "campo a campo" en cada atributo de ella; luego, esta clase fue incluida en todos los archivos donde se utilicen variables de tipo Muestra.

Luego en el proceso de adquisición de los datos se lee el archivo con los datos generados por el proceso anterior, se procesa esa información leyendo de a una línea y luego dividiéndola en cada guión (todo esto se realiza con la ayuda de las funciones provistas por librerías como "os" y "string", las cuales son importadas al comienzo de cada archivo que las requiera), y una vez obtenidos los valores de los parámetros se los muestra a través de la página web (el sitio web es creado utilizando Flask, como se explicó anteriormente en el inciso 1). Se decidió que al tomar una muestra esta no sea eliminada del archivo de datos; esto es para dejar registro de los datos obtenidos para una posible extensión del sistema donde se desee calcular algunas estadísticas.

Cabe aclarar que debido a la forma en que mostramos los datos en la página web, que es refrescando la página web, en lugar de hacerlo por defecto cada un segundo, decidimos dejar cada cinco segundos el refresco automático, que es la cantidad de segundos de la frecuencia de muestreo; cada vez que se refresca la página estarán las últimas muestras captados por los sensores.

El único procesamiento que se lleva a cabo de la información es la de sacar el promedio de las últimas 10 muestras de cada sensor. El promedio se va llevando a cabo con las últimas 10 muestras de cada sensor. Se van guardando las últimas 10 muestras en un arreglo, se suman, y luego se divide por la dimensión misma del arreglo. El arreglo en realidad funciona como si fuera un buffer, el cual va guardando el contenido de las últimas 10 muestras, y una vez que se llena se continua por el principio del arreglo. Hay que aclarar que para las primeras muestras que se muestran, cuando todavía no llegan a ser 10, el promedio que se hace es a partir de las muestras que se van mostrando y la dimensión en ese momento del arreglo. De esta forma, por ejemplo, cuando recién se muestran las dos

primeras muestras, el promedio que se muestra es la suma de estas dos muestras dividida por dos, y así sucesivamente hasta llegar a las 10.

La única interacción por parte del usuario es el acceder a la página, que eso comienza con el proceso de adquisición y muestreo de los datos.

4) Los cambios de HTTP que cambian en este caso sería que se agrega una respuesta con el método POST debido a que el usuario elige la frecuencia de muestreo con la que se mostrarán los datos. La atención de éste método es causada únicamente cuando el usuario manda información mediante la página web gracias al *action_form*, que a nivel HTTP le indica al servidor el tipo de petición que el usuario requiere. La atención del método POST en nuestro código solamente guarda la nueva frecuencia de muestreo en una variable y luego se vuelve a la ejecución del método principal (*index()*) con la nueva frecuencia de muestreo.

A nivel html sólo cambiaría el valor de la frecuencia de muestreo debido a que no se cambia a otra vista.

A nivel de la simulación, como ya explicamos en el inciso anterior el tema del refresco automático de la página, y como en este inciso se agrega la posibilidad de cambiar la frecuencia de muestreo, debido a esta instancia de el usuario poder cambiar la frecuencia de muestreo, decidimos que el muestreo por defecto sean de cinco segundos. Esto es debido a que si la frecuencia de muestreo de los datos es pequeña, por ejemplo un segundo o dos inclusive, el usuario no podrá cambiar la frecuencia debido a que la página se refrescará cada un segundo, y esto no daría tiempo al usuario a querer cambiar la frecuencia si lo quisiera.

Esta es la información capturada desde RawCap de la misma forma que en el inciso anterior cuando se realiza el POST:

```
Wireshark · Packet 54 · localhostcapture2

Raw packet data
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
> Transmission Control Protocol, Src Port: 63892, Dst Port: 8765, Seq: 1, Ack: 1, Len: 599
  Hypertext Transfer Protocol
    POST / HTTP/1.1\r\n
    [Expert Info (Chat/Sequence): POST / HTTP/1.1\r\n]
    Request Method: POST
    Request URI: /
    Request Version: HTTP/1.1
    Host: localhost:8765\r\n
    Connection: keep-alive\r\n
    Content-Length: 12\r\n
    Cache-Control: max-age=0\r\n
    Origin: http://localhost:8765\r\n
    Upgrade-Insecure-Requests: 1\r\n
    Content-Type: application/x-www-form-urlencoded\r\n
    User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/61.0.3163.100 Safari/537.36\r\n
    Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8\r\n
    Referer: http://localhost:8765/\r\n
    Accept-Encoding: gzip, deflate, br\r\n
    Accept-Language: en-US,en;q=0.8,es-419;q=0.6,es;q=0.4,tr;q=0.2,gl;q=0.2\r\n
    \r\n
    [Full request URI: http://localhost:8765/]
    [HTTP request 1/1]
    [Response in frame: 70]
    File Data: 12 bytes
  HTML Form URL Encoded: application/x-www-form-urlencoded

01e0 38 0d 0a 52 65 66 65 72 65 72 3a 20 68 74 74 70 8..Refer er: http
01f0 3a 2f 2f 6c 6f 63 61 6c 68 6f 73 74 3a 38 37 36 ://local host:876
0200 35 2f 0d 0a 41 63 63 65 70 74 2d 45 6e 63 6f 64 5/..Acce pt-encod
0210 69 6e 67 3a 20 67 7a 69 70 2c 20 61 65 66 6c 61 ing: gzi p, defla
0220 74 65 2c 20 62 72 0d 0a 41 63 63 65 70 74 2d 4c te, br.. Accept-L
0230 61 6e 67 75 61 67 65 3a 20 65 6e 2d 55 53 2c 65 anguage: en-US,e
0240 6e 3b 71 3d 30 2e 38 2c 65 73 2d 34 31 39 3b 71 n;q=0.8, es-419;q
0250 3d 30 2e 36 2c 65 73 3b 71 3d 30 2e 34 2c 74 72 =0.6,es; q=0.4,tr
0260 3b 71 3d 30 2e 32 2c 67 6c 3b 71 3d 30 2e 32 0d ;q=0.2,gl;q=0.2.
0270 0a 0d 0a 66 72 65 63 75 65 6e 63 69 61 3d 37 ...frecu encia=7
```

Y esta es la “conversación” entre el cliente y el servidor:


```

POST / HTTP/1.1
Host: localhost:8765
Connection: keep-alive
Content-Length: 13
Cache-Control: max-age=0
Origin: http://localhost:8765
Upgrade-Insecure-Requests: 1
Content-Type: application/x-www-form-urlencoded
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/61.0.3163.100 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8
Referer: http://localhost:8765/
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.8,es-419;q=0.6,es;q=0.4,tr;q=0.2,gl;q=0.2

frecuencia=15 HTTP/1.0 200 OK
Content-type: text/html; charset=utf-8
Content-Length: 3616
Server: Werkzeug/0.12.2 Python/2.7.6
Date: Thu, 05 Oct 2017 21:08:49 GMT

<html>
<head>
  <meta name="viewport" content="width=device-width, initial-scale=1">

```

En **rojo** se muestran los paquetes mandados por el cliente, y la respuesta en **azul** del servidor. Encuadramos en **naranja** la información que envía el usuario para el cambio de la frecuencia de muestreo.

5) El mayor problema de concurrencia se encuentra en la lectura y escritura en el archivo de texto que funciona como “base de datos”, dado que solamente un proceso debe poder acceder al archivo en un determinado instante. En nuestra simulación no ocurrieron estos errores gracias a la velocidad del procesador de la computadora en la cual se efectúan estas acciones.

Particularmente en el momento que se cambia la frecuencia de muestreo, como se llama a un método POST para recibir la información y luego se utiliza nuevamente la función *render_template()* dentro del código; lo que sucede acto seguido de cargar la misma página luego de enviarse la nueva frecuencia de muestreo, es que se cambian las muestras en ese instante debido al refresco de la página, luego se muestran los datos en el período requerido. Para ejemplificar y aclarar esta situación: Una vez que entramos en la página para ver los datos, éstos se van mostrando cada cinco segundos por defecto. Luego el usuario tiene la opción de elegir la frecuencia que él desee que quiere ver los datos. En el momento en el que ingresa por ejemplo diez segundos, en el instante en el que se aprieta el botón para enviar la información del cambio de la frecuencia, debido a que el método POST responde a ésta petición del usuario, y en la forma en la que nosotros atendemos esta petición en el programa, que es renderizar la misma página, pero con la frecuencia de muestreo cambiada, en el momento que se renderiza nuevamente la página, ya están dispuestos nuevos datos y desde ese momento empieza la cuenta de la nueva frecuencia de muestreo. De esta forma cuando el usuario cambia la frecuencia, no se espera a mostrar los siguientes datos que venían con la “vieja” frecuencia de muestreo.

Si no se cuenta con las medidas adecuadas de prevención en la concurrencia de los procesos, sí se producirían problemas difíciles de simular debido a que se puede cambiar la frecuencia de muestreo de alguna medida en un instante en dónde se debería de leer en lugar de escribir.

Problemas de tiempo real que podrían producirse:

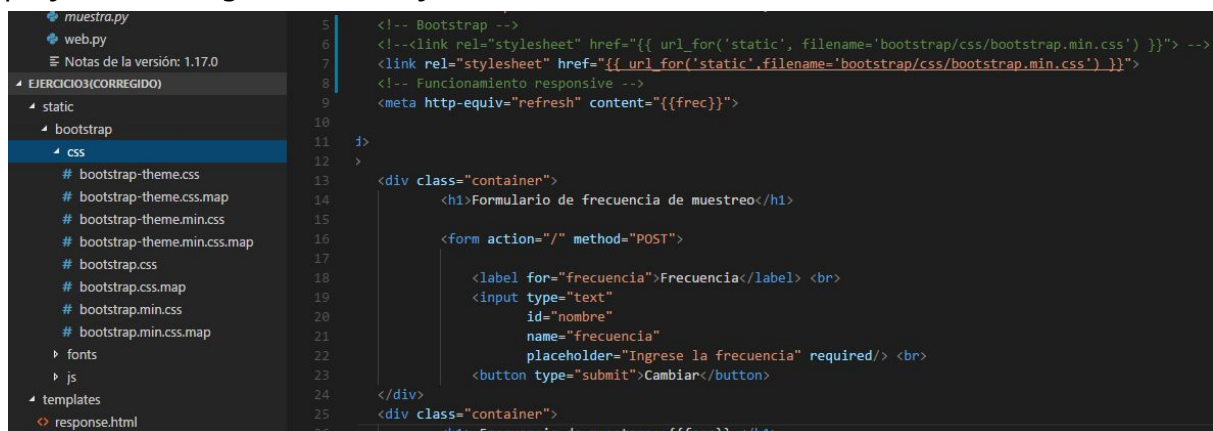
- Generación de los datos fuera del rango de tiempo en que se los debe generar.

- Lectura de un dato que todavía no se haya generado o actualizado.
- Que los datos no se generen a la frecuencia de muestreo deseada.

6) Principalmente debido a que la simulación no está sujeta a errores de aproximación o errores que se deban a causas físicas (como ruido, por ejemplo). Otro de los problemas fundamentales será que seguramente los valores sean adquiridos a través de cuatro sensores independientes, por lo cual se tendrá que tener en cuenta cuanto tiempo se demora realmente en tomar una muestra compuesta por las mediciones de los cuatro sensores. Los sensores reales además están sujetos a problemas físicos, recalibración, alimentación y demás, los cuáles se abstraen completamente en la simulación planteada. Luego pueden darse errores en las mediciones reales que uno antes de mostrar debe de estar seguro si puede ser un valor real, o tomarlo en cuenta como alerta por valor atípico por ejemplo. Se debe tener en cuenta en cuenta la conexión con la base de datos que esté en un estado correcto a la hora de leer los datos de la misma.

Adjunto: Error de Bootstrap

En la entrega anterior del Trabajo Práctico 1 el funcionamiento del ejercicio resuelto por nosotros es exactamente igual al de esta re-entrega. El único detalle es que anteriormente no pudimos hacer funcionar Bootstrap en nuestro proyecto. Lo habíamos agregado al proyecto de la siguiente forma y no funcionaba:



```

5 <!-- Bootstrap -->
6 <!--<link rel="stylesheet" href="{{ url_for('static', filename='bootstrap/css/bootstrap.min.css') }}" -->
7 <link rel="stylesheet" href="{{ url_for('static', filename='bootstrap/css/bootstrap.min.css') }}">
8 <!-- Funcionamiento responsive -->
9 <meta http-equiv="refresh" content="{{frec}}">
10
11 </div>
12 </div>
13 <div class="container">
14 <h1>Formulario de frecuencia de muestreo</h1>
15
16 <form action="/" method="POST">
17
18 <label for="frecuencia">Frecuencia</label> <br>
19 <input type="text"
20 <div class="form-group">
21 <input type="text"
22 <input type="text"
23 <input type="text"
24 </div>
25 <div class="container">
26 <div class="form-group">

```

En este momento lo agregamos de forma CDN y funciona sin problemas, de esta forma logramos un sitio web responsive.

Acá adjunto el error por consola que nos aparecía anteriormente:


```

Jorch@LAPTOP-LUD1N7NR MINGW64 ~/Desktop/Taller de Proyecto II/Práctica/Ejercicio
3(corregido) (master)
$ python web.py
* Running on http://0.0.0.0:8765/ (Press CTRL+C to quit)
127.0.0.1 - - [06/Oct/2017 01:12:40] "GET / HTTP/1.1" 200 -
[2017-10-06 01:12:40,855] ERROR in app: Exception on /static/bootstrap/css/bootstrap.min.css [GET]
Traceback (most recent call last):
  File "C:\Python27\lib\site-packages\flask\app.py", line 1982, in wsgi_app
    response = self.full_dispatch_request()
  File "C:\Python27\lib\site-packages\flask\app.py", line 1614, in full_dispatch_request
    rv = self.handle_user_exception(e)
  File "C:\Python27\lib\site-packages\flask\app.py", line 1517, in handle_user_exception
    reraise(exc_type, exc_value, tb)
  File "C:\Python27\lib\site-packages\flask\app.py", line 1612, in full_dispatch_request
    rv = self.dispatch_request()
  File "C:\Python27\lib\site-packages\flask\app.py", line 1598, in dispatch_request
    return self.view_functions[rule.endpoint](**req.view_args)
  File "C:\Python27\lib\site-packages\flask\helpers.py", line 928, in send_static_file
    cache_timeout=cache_timeout)
  File "C:\Python27\lib\site-packages\flask\helpers.py", line 669, in send_from_directory
    filename = safe_join(directory, filename)
  File "C:\Python27\lib\site-packages\flask\helpers.py", line 639, in safe_join
    return posixpath.join(*parts)
  File "C:\Python27\lib\posixpath.py", line 80, in join
    path += '/' + b
UnicodeDecodeError: 'ascii' codec can't decode byte 0xe1 in position 47: ordinal not in range(128)
127.0.0.1 - - [06/Oct/2017 01:12:40] "GET /static/bootstrap/css/bootstrap.min.css HTTP/1.1" 500 -
127.0.0.1 - - [06/Oct/2017 01:12:46] "GET / HTTP/1.1" 200 -

```