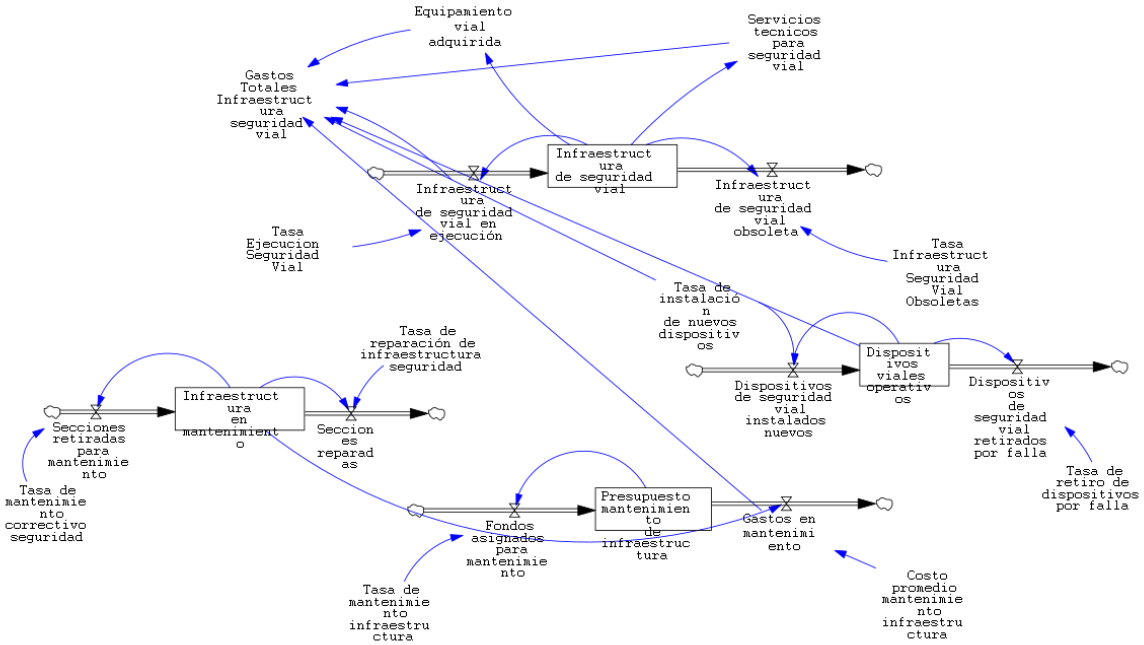
MANUAL VENSIM TO PYTHON (WEB)

Índice

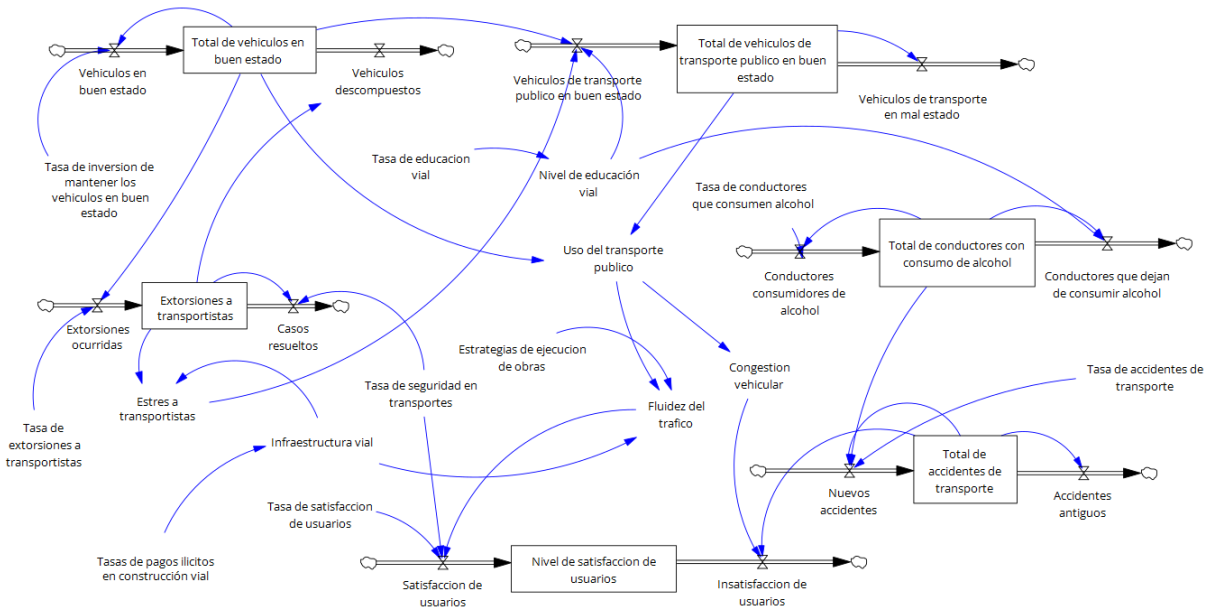
1. Análisis

1.1. Modelos Vensim

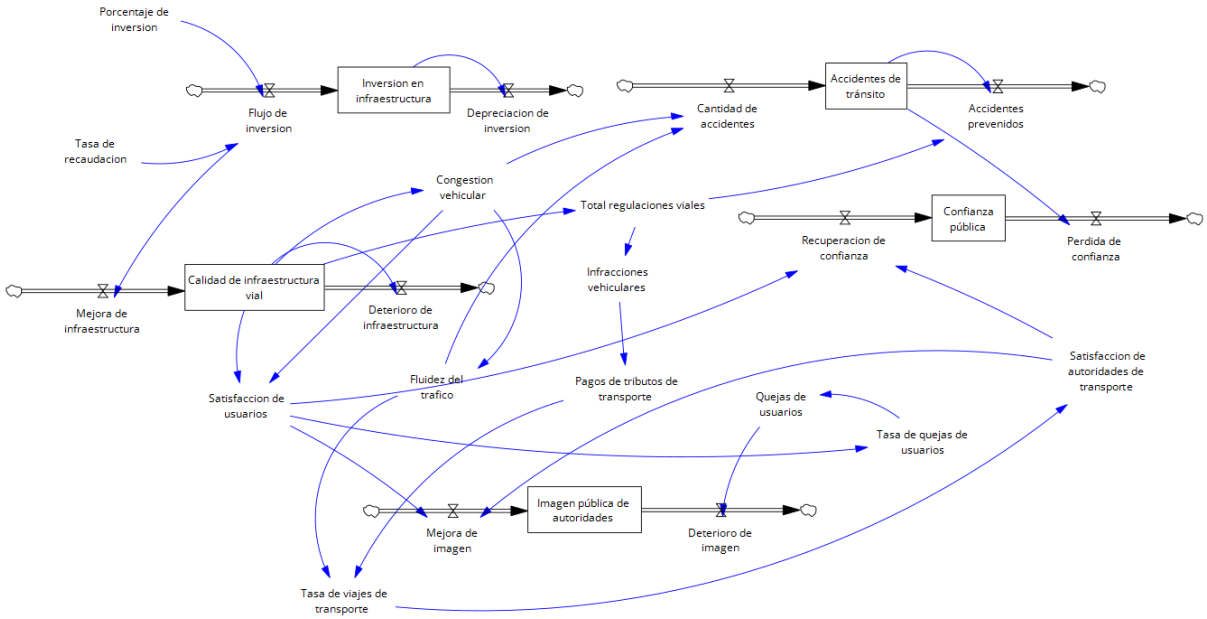
Seguridad Vial



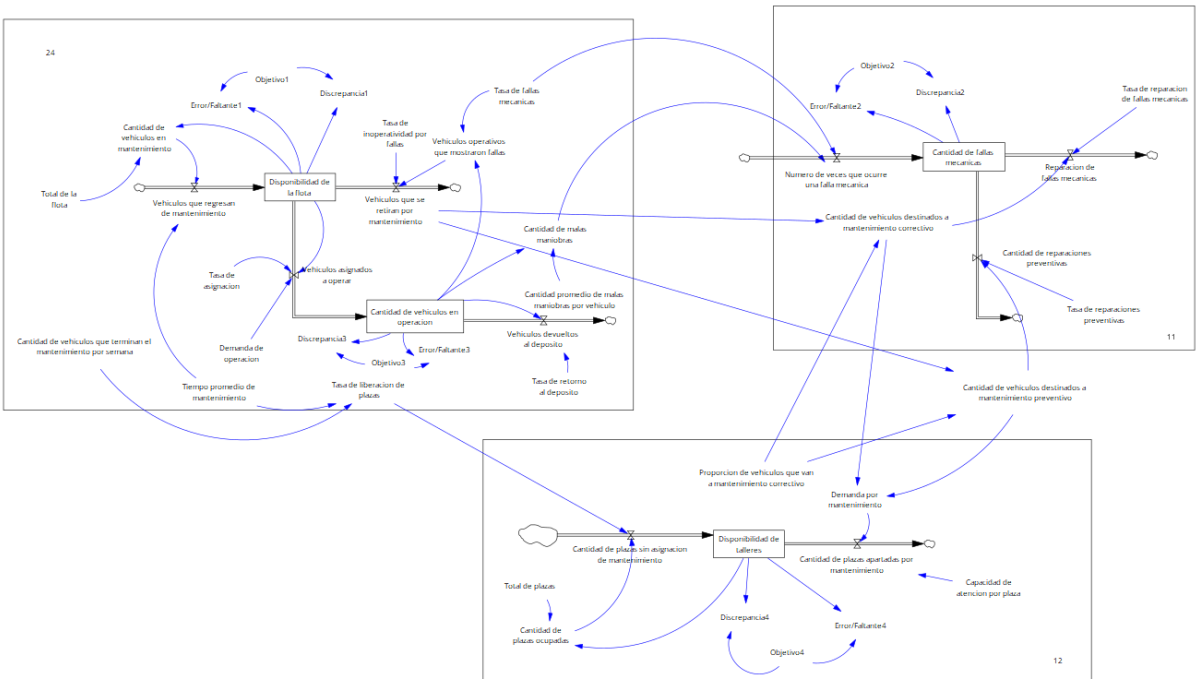
Satisfacción de usuarios



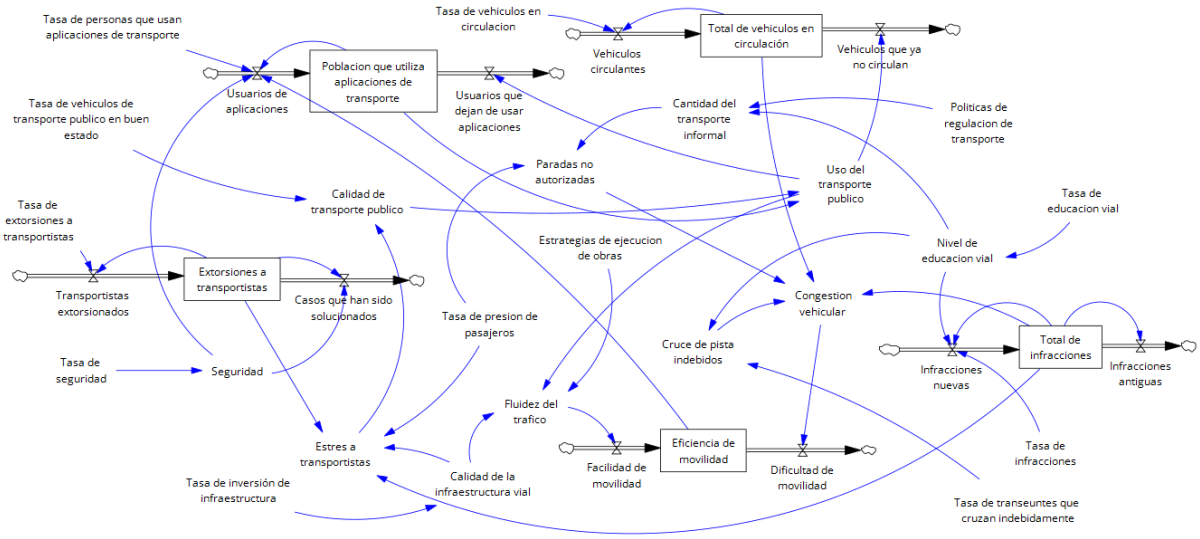
Satisfacción de autoridades



Frecuencia de mantenimiento



Eficiencia de movilidad



Estos modelos nos servirán como base para poder llevarlo a un entorno Web usando Python con la librería Flask.

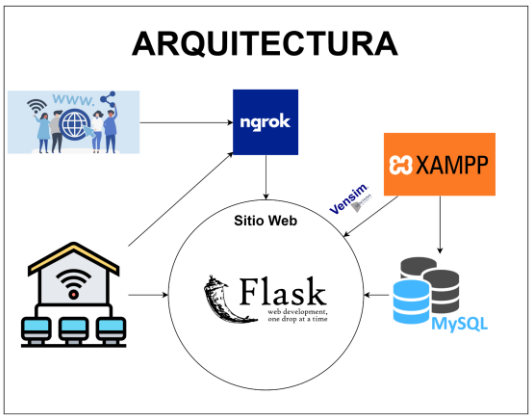
1.2. Requisitos Funcionales

#	Descripción
RF1	Página web responsiva para tamaños de pantalla (Tablet, Laptop, Monitor, Celular)
RF2	Mostrar en web las gráficas del Vensim (mdl) en base a su Nivel (sin límite)
RF3	Configuración de gráficas mostradas del vensim debe ser desde (título, nombre de eje x, nombre de eje y, color de gráfica, nombre del nivel y poder modificar posición mostrada en web)
RF4	Controlar errores con una pantalla visual (error de servidor no disponible, mal configurado variables de entorno, nivel mal escrito en BD o no encontrar el archivo mdl en servidor)

1.3. Requisitos no Funcionales

#	Descripción
RNF1	Página web responsiva para tamaños de pantalla (Tablet, Laptop, Monitor, Celular)

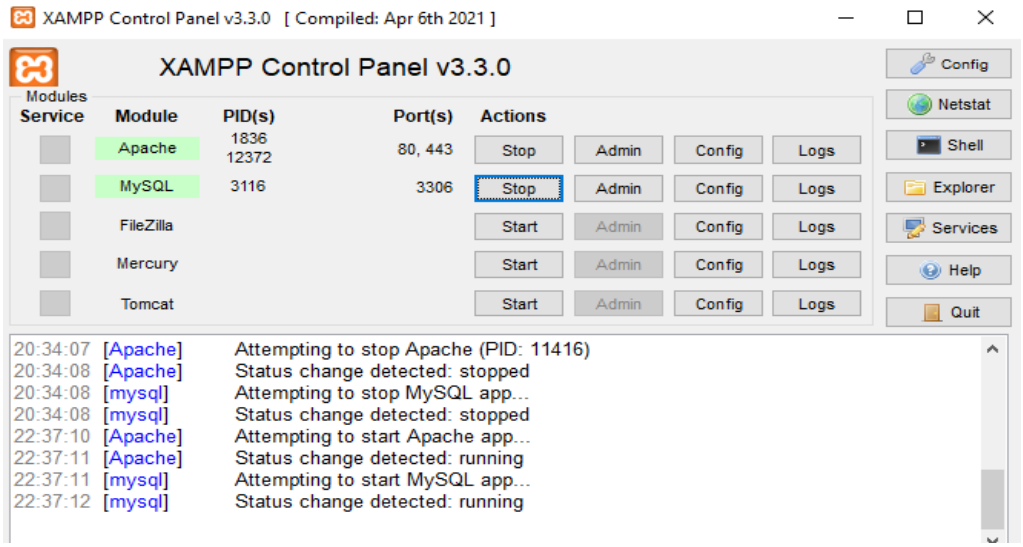
1.4. Arquitectura de Desarrollo



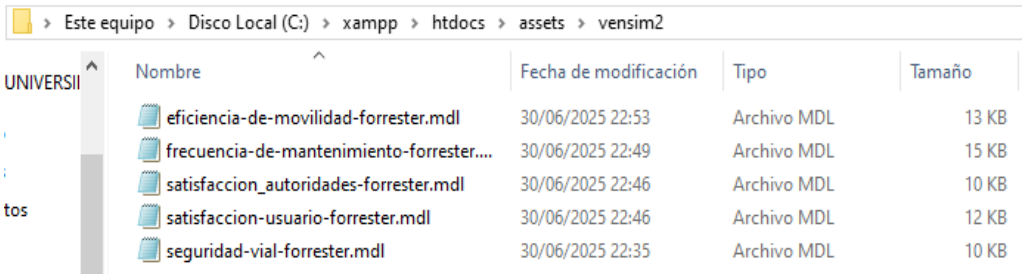
2. Desarrollo

2.1. Instalaciones Previas

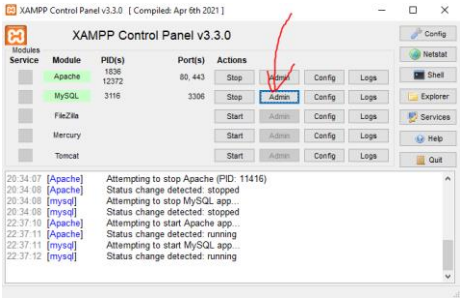
- Tener tu servidor local Xampp instalado e iniciando los servicios de Apache y Mysql. Se puede descargar e instalar Xampp desde: https://www.apachefriends.org/es/download.html o instalar a través del ejecutable que se encuentra en la carpeta Instalaciones.

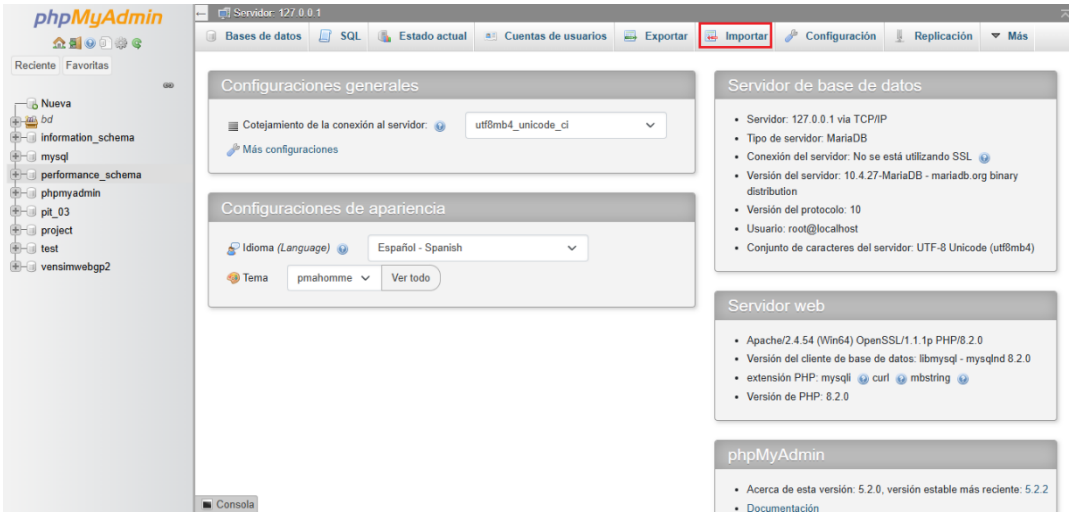


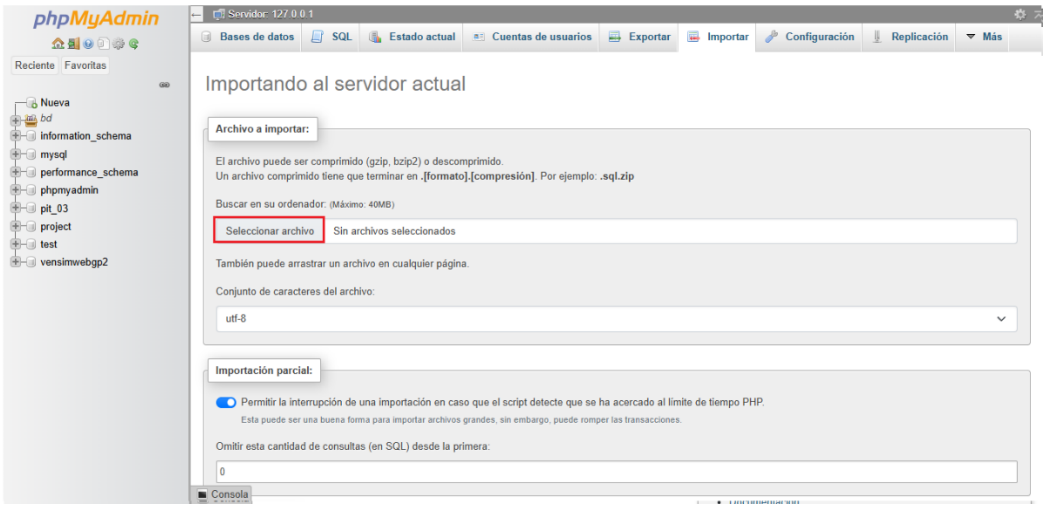
- Dentro del directorio público de servidor local crear las carpetas para tener la siguiente ruta (C:\xampp\htdocs\assets\vensim2)
- Dentro de la ruta creada colocar los modelos vensim (se encuentran en la carpeta Backup)

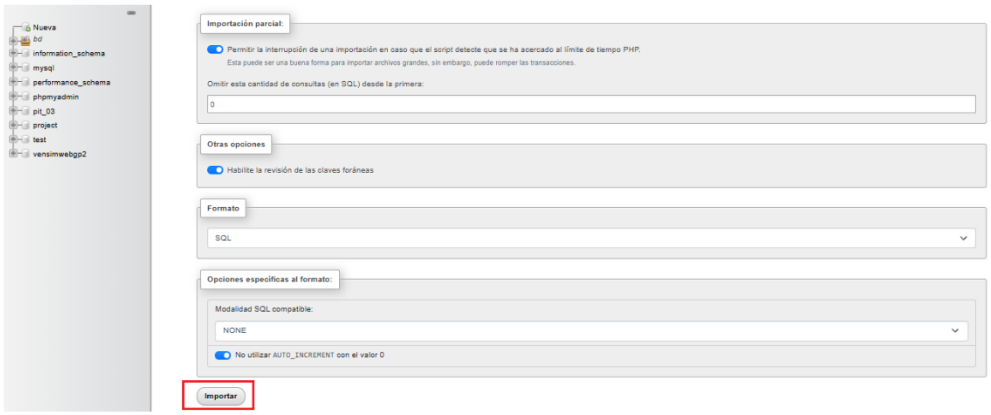


- Abrir Admin del Módulo de MySQL dentro de Xampp y e importar el archivo vensimwebGp2.sql (el archivo se encuentra en Backup), este archivo tendrá todo lo necesario para no requerir cambios en el documento .env









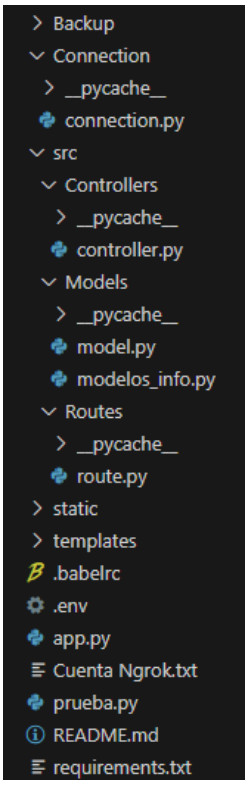
- Si configuras de distinta manera actualizar el .env

```
.env
APP_NAME=Vensim
APP_URL_VENSIM=http://localhost/assets/vensim2/

DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=vensimwebgp2
DB_USERNAME=root
DB_PASSWORD=
```

- Instalar las siguientes librerías de Python:
  - Flask (pip install flask)
  - Mysql (pip install mysql-connector-python)
  - Decouple (pip install python-decouple)
  - urllib3 (pip install urllib3)
  - mpld3 (pip install mpld3)
  - pyngrok (pip install pyngrok)
  - ssl (pip install ssl)

2.2. Desarrollo del Código

Arquitectura de Código Fuente



Utilizaremos una arquitectura llamada MVC (Modelo-Vista-Controlador). Sin embargo, en este caso, hay una pequeña variación llamada "Route".

1. Comenzamos con un archivo llamado app.py. Piensa en él como el punto de partida de tu aplicación. Aquí configuras todo y preparas el terreno.
2. Luego, tenemos un archivo llamado route.py. Este archivo se encarga de manejar las diferentes rutas o direcciones URL a las que los usuarios pueden acceder en tu aplicación. Cuando un usuario solicita una URL específica, route.py se activa y redirige la solicitud a la función adecuada en el archivo controller.py.
3. En el archivo controller.py, es donde se encuentra toda la lógica del negocio. Aquí es donde procesas la solicitud del usuario, realizas cálculos, consultas a la base de datos y cualquier otra operación necesaria. También puedes descargar información de una herramienta llamada Vensim para realizar cálculos más avanzados.
4. Para conectarte a la base de datos, utilizas un archivo llamado connection.py. Este archivo se encarga de establecer la conexión con la base de datos y proporciona métodos para realizar consultas y actualizaciones en la misma. El archivo model.py se comunica con connection.py para hacer consultas a la base de datos y obtener los datos necesarios para la lógica del negocio en controller.py.
5. Después de realizar todas las operaciones necesarias en controller.py, devuelves los resultados a route.py. Aquí puedes decidir si mostrar una plantilla HTML (como template.html) con los resultados procesados para el usuario, o si ocurrió algún problema, mostrar una página de error (como error.html).

Main (app.py)

```python
import ssl
from flask import Flask
from src.Routes.route import modelRoute

from pyngrok import ngrok
ssl._create_default_https_context = ssl._create_unverified_context
ngrok.set_auth_token('2hKn4UWWXYkrNTCBS8HWz0IKXtY_6YwXJMKDn4w3KWwTZUFfN')
app = Flask(__name__)
modelRoute(app)

if __name__ == "__main__":
    public_url = ngrok.connect(5000)
    # Obtener la URL pública generada por Ngrok
    public_url_str = str(public_url)
    print('URL pública de Ngrok:', public_url_str)
    app.run()
```

Se utiliza Flask para la creación Web, ngrok para el túnel, agregando el API para la conexión obtenido de su cuenta https://ngrok.com/

Route (route.py)

```python
from flask import render_template,make_response
from src.Controllers.controller import controller

def modelRoute(app):
    @app.route('/', methods=['GET'])
    def model():
        response = controller()
        #print(response)
        if not (isinstance(response, dict) and len(response) == 0):
            if not (isinstance(response, list) and 'message' in response[0]):
                respuesta = make_response(render_template('template.html', nivel=response))
                respuesta.headers['Cache-Control'] = 'public, max-age=180'
                respuesta.headers['X-Content-Type-Options'] = 'nosniff'
                respuesta.headers['Server'] = 'Nombre del servidor'
                return respuesta
            else:
                respuesta = make_response(render_template('error.html', error_message=response))
                return respuesta
        else:
            respuesta = make_response(render_template('error.html', error_message=response))
            return respuesta
```

make_response: Se utiliza para crear una respuesta HTTP personalizada. Puedes utilizar esta función para construir una respuesta con contenido personalizado, encabezados especiales u otros ajustes específicos. En el código proporcionado, se utiliza make_response para crear una respuesta HTTP a partir de la plantilla renderizada.

render_template: Se utiliza para renderizar una plantilla HTML utilizando el motor de plantillas de Flask. Esta función toma el nombre de la plantilla como argumento y devuelve el contenido HTML generado, se utiliza render_template para generar el contenido HTML a partir de la plantilla 'template.html' o 'error.html', dependiendo de la respuesta recibida del controlador.

controller: Se está realizando el llamado al controlador que contiene la lógica desarrollada

Controllers (controller.py)

```python
from src.Models.model import getModelAll
import pysd
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
import urllib3
import os
import mpld3
from decouple import config

def controller():
    carpeta_destino = 'static/vensim/modelos'
    nivel = {}
    response = getModelAll()

    archivos_mdl = [
        'frecuencia-de-mantenimiento-forrester.mdl',
        'eficiencia-de-movilidad-forrester.mdl',
        'satisfaccion_autoridades-forrester.mdl',
        'satisfaccion-usuario-forrester.mdl',
        'seguridad-vial-forrester.mdl'
    ]

    if not (isinstance(response, list) and 'message' in response[0]):
        response_format = [{
            "idModel": str(item[0]),
            "nameModel": item[1],
            "idSubmodel": str(item[2]),
            "title": item[3],
            "nameLabelX": item[4],
            "nameLabelY": item[5],
            "position": item[6],
            "nameSubmodel": item[7],
            "nameColor": item[8],
            "nameNivel": item[7],
        } for item in response]

        print("[DEBUG] Diccionarios cargados desde la base de datos:")
        for i, item in enumerate(response_format, start=1):
            print(f"[DEBUG] Diccionario {i}: {item}")

        try:
            os.makedirs(carpeta_destino, exist_ok=True)
            url_base = config('APP_URL_VENSIM')
            http = urllib3.PoolManager()

            for nombre_archivo in archivos_mdl:
                ruta_archivo = os.path.join(carpeta_destino, nombre_archivo)
                url_modelo = url_base + nombre_archivo
                print(f"[DEBUG] Descargando {url_modelo} → {ruta_archivo}")

                try:
                    resp = http.request('GET', url_modelo)
                    with open(ruta_archivo, 'wb') as archivo:
                        archivo.write(resp.data)
                except Exception as e:
                    return [{'message': f'Error descargando {nombre_archivo}: {str(e)}'}]

            for nombre_archivo in archivos_mdl:
                ruta_archivo = os.path.join(carpeta_destino, nombre_archivo)
```

```
        for nombre_archivo in archivos_mdl:
            ruta_archivo = os.path.join(carpeta_destino, nombre_archivo)

            try:
                model = pysd.read_vensim(ruta_archivo)
            except Exception as e:
                return [{'message': f'Error leyendo el archivo {nombre_archivo}: {str(e)}'}]

            try:
                stocks = model.run()
            except Exception as e:
                return [{'message': f'Error ejecutando simulación del archivo {nombre_archivo}: {str(e)}'}]

            for i in response_format:
                try:
                    if i['nameNivel'] not in stocks.columns:
                        print(f"[DEBUG] Nivel '{i['nameNivel']}' no está en {nombre_archivo}")
                        continue

                    stock_data = stocks[i['nameNivel']].head(10).to_dict()

                    plt.plot(stocks[i['nameNivel']], label=i['nameNivel'],
                             linewidth=4.0, color=i['nameColor'])
                    plt.title(i['title'], loc='center')
                    plt.ylabel(i['nameLabelY'])
                    plt.xlabel(i['nameLabelX'])
                    plt.grid()
                    plt.legend(loc='center left', facecolor='black',
                               framealpha=1.0, edgecolor='black',
                               labelcolor='white')
                    plt_graph = mpld3.fig_to_html(plt.gcf())
                    plt.close()

                    modelo = i['nameModel']
                    submodelo = i['nameSubmodel']

                    if modelo not in nivel:
                        nivel[modelo] = {}

                    nivel[modelo][submodelo] = {
                        'data': stock_data,
                        'graph': plt_graph,
                        'title': i['title'],
                        'ylabel': i['nameLabelY'],
                        'xlabel': i['nameLabelX']
                    }

                    print(f"[DEBUG] Gráfico generado para '{i['nameNivel']}' en {nombre_archivo}")

                except Exception as e:
                    return [{'message': f"Error graficando variable {i['nameNivel']} desde archivo {nombre_archivo}: {str(e)}"}]

        return nivel

    except Exception as e:
        return [{'message': f'Error general en controller: {str(e)}'}]

else:
    return response
```

Importaciones de bibliotecas:

- getModelAll: Es una función importada desde src.Models.model que se utiliza para obtener los datos del modelo desde la base de datos.
- pysd: Es una biblioteca que permite leer y simular modelos Vensim.
- matplotlib es una biblioteca para visualización de datos y gráficos.
- urllib3: Es una biblioteca utilizada para realizar solicitudes HTTP.
- os: Es una biblioteca para interactuar con el sistema operativo, en este caso, para manejar rutas de archivos.
- mpld3: Es una biblioteca que permite convertir gráficos de Matplotlib en HTML interactivo.
- decouple es una biblioteca utilizada para cargar variables de entorno desde un archivo de configuración externo.

Interacción:

La función controller() es el controlador principal que realiza el procesamiento y la lógica del negocio. Primero, se define una carpeta de destino y un nombre de archivo para la descarga del archivo Vensim. Luego, se llama a la función getModelAll() para obtener los datos del modelo desde la base de datos.Si la respuesta de la base de datos no contiene un mensaje de error, se realiza lo siguiente:

- Los datos de la base de datos se convierten en un formato específico.
- Se descarga el archivo Vensim desde una URL especificada y se guarda en una carpeta local.

- Se intenta leer y simular el modelo Vensim utilizando pysd.
- Para cada nivel en los datos del modelo:
    - Se ejecuta el modelo y se extraen los datos de los stocks.
    - Se crea un gráfico utilizando matplotlib con etiquetas y estilos personalizados.
    - Se convierte el gráfico en HTML interactivo utilizando mpld3.
    - Los datos y el gráfico se almacenan en un diccionario llamado nivel.
- Finalmente, se devuelve el diccionario nivel con los datos y gráficos generados.

Si ocurre algún error en la lectura del archivo Vensim o en la simulación del modelo, se devuelve un mensaje de error correspondiente.

Models (model.py)

```python
from Connection.connection import connect, connection_select
from src.Models.modelos_info import modelos_info  # Diccionario con estructura lógica

def getModelAll():
    dataBase = connect()
    if not isinstance(dataBase, list):
        cursorObject = dataBase.cursor()
        try:
            stmt = (
                "SELECT "
                "m.idModel, "
                "m.nameModel, "
                "s.idSubmodel, "
                "s.title, "
                "s.nameLabelX, "
                "s.nameLabelY, "
                "s.position, "
                "s.nameSubmodel, "          # <- esta es la variable a graficar
                "c.nameColor "
                "FROM submodel s "
                "JOIN model m ON s.idModel = m.idModel "
                "JOIN color c ON s.position = c.idColor "
                "ORDER BY s.position;"
            )
            myresult = connection_select(cursorObject, stmt)
            cursorObject.close()
            dataBase.close()
            return myresult
        except Exception as e:
            cursorObject.close()
            dataBase.close()
            return [{'message': f'Error en consulta getModelAll: {str(e)}'}]
    else:
        error_message = dataBase[0]['message']
        return [{'message': error_message}]


def getModeloInfo():
    """
    Retorna el diccionario que define qué variables
    se deben graficar por modelo y submodelo.
    """
    return modelos_info
```

Importaciones de bibliotecas:

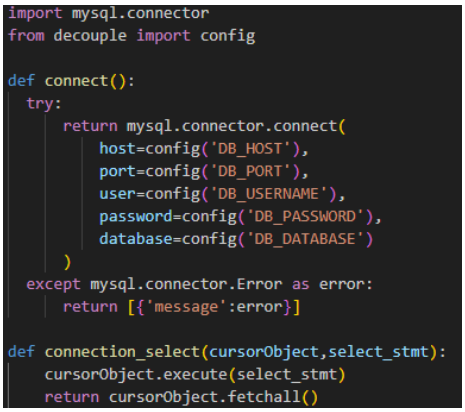
- **connect:** Es una función importada desde Connection.connection que se utiliza para establecer la conexión con la base de datos.
- **connection_select:** Es una función importada desde Connection.connection que se utiliza para realizar una consulta select en la base de datos.

Función getModelAll():

- La función getModelAll() se encarga de obtener todos los parámetros del vensim desde la base de datos.
- Primero, se establece una conexión con la base de datos utilizando la función connect().
- Si la conexión con la base de datos no es una lista (lo que indica un error de conexión), se procede a ejecutar una consulta select en la base de datos.
- La consulta SQL selecciona varios campos de la tabla 'model' y 'color' y los une en

base a una relación entre las tablas.
- Se ejecuta la consulta utilizando la función connection_select() y se guarda el resultado en myresult.
- Luego, se cierra el cursor y se cierra la conexión con la base de datos.
- Si todo se realiza correctamente, se devuelve el resultado de la consulta myresult.
- Si ocurre algún error durante la ejecución de la consulta, se devuelve un mensaje de error correspondiente.

Connection (connection.py)



```python
import mysql.connector
from decouple import config

def connect():
  try:
      return mysql.connector.connect(
          host=config('DB_HOST'),
          port=config('DB_PORT'),
          user=config('DB_USERNAME'),
          password=config('DB_PASSWORD'),
          database=config('DB_DATABASE')
      )
  except mysql.connector.Error as error:
      return [{'message':error}]

def connection_select(cursorObject,select_stmt):
    cursorObject.execute(select_stmt)
    return cursorObject.fetchall()
```

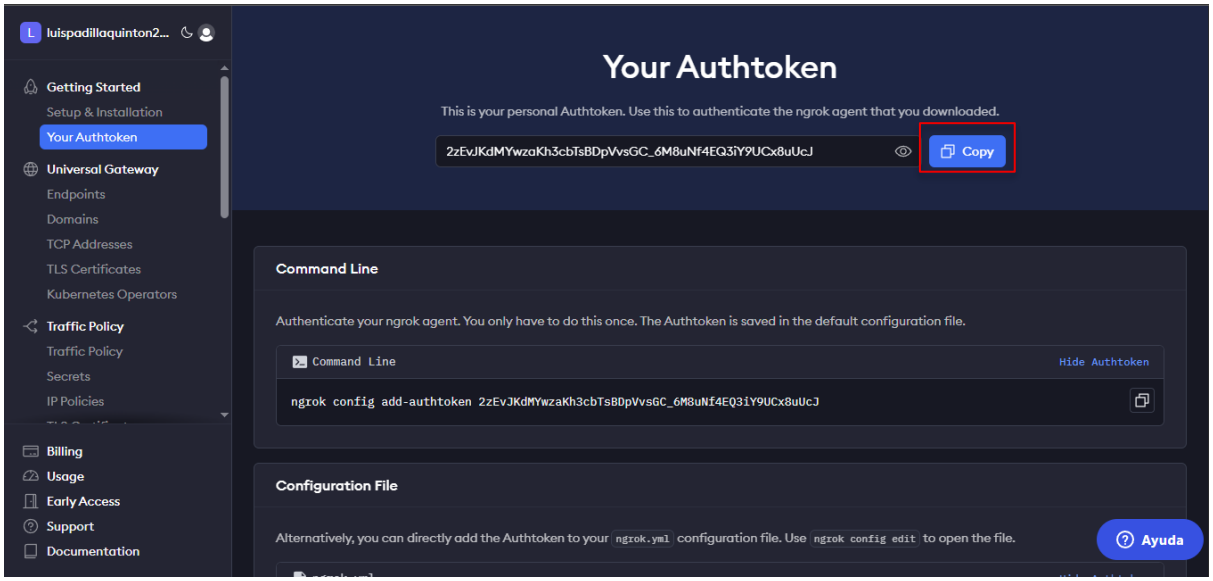
Importaciones de bibliotecas:
- **mysql.connector:** Es una biblioteca utilizada para conectarse a la base de datos MySQL.
- **decouple:** Es una biblioteca utilizada para cargar variables de entorno desde un archivo de configuración externo.

Función connect():
- La función connect() se encarga de establecer la conexión con la base de datos MySQL.
- Utiliza los datos de configuración proporcionados en las variables de entorno (como el host, puerto, nombre de usuario, contraseña y base de datos) utilizando la función config() de la biblioteca decouple.
- Intenta establecer la conexión utilizando la biblioteca mysql.connector.connect().
- Si la conexión se establece correctamente, se devuelve el objeto de conexión.
- Si ocurre algún error durante la conexión, se captura la excepción mysql.connector.Error y se devuelve un diccionario con un mensaje de error correspondiente.

Función connection_select(cursorObject, select_stmt):
- La función connection_select() se encarga de ejecutar una consulta select en la base de datos utilizando un objeto de cursor y una sentencia select proporcionada.
- Utiliza el objeto de cursor proporcionado para ejecutar la consulta select utilizando el método execute() del cursor.
- Luego, utiliza el método fetchall() del cursor para obtener todos los resultados de la consulta.
- Finalmente, devuelve los resultados obtenidos.

3. Prueba

Si todo fue realizado correctamente. Crear una cuenta de Ngrok (o usar la cuenta proporcionada en el archivo Cuenta Ngrok.txt), iniciar sesión y copiar el Authtoken. (Revisar el archivo Pasos Ngrok.txt para más detalles)



Pegar el authoken en app.py y ejecutar

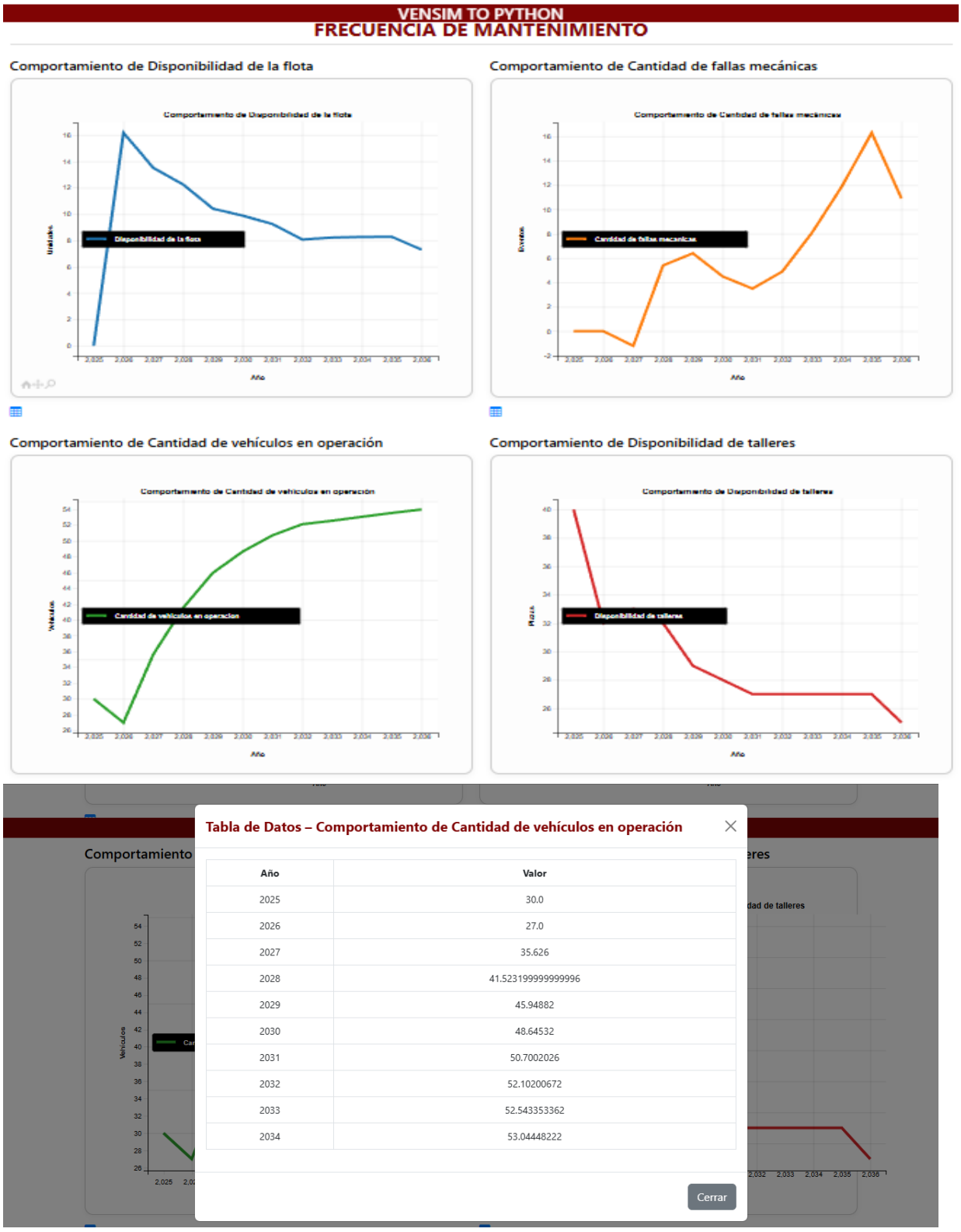
app.py - C:\Users\Usser\Downloads\4PC\project\app.py (3.9.4)

File Edit Format Run Options Window Help

```
import ssl
from flask import Flask
from src.Routes.route import modelRoute

from pyngrok import ngrok
ssl._create_default_https_context = ssl._create_unverified_context
ngrok.set_auth_token('2zEvJKdMYwzaKh3cbTsBDpVvsGC_6M8uNf4EQ3iY9UCx8uUcJ')
app = Flask(__name__)
modelRoute(app)

if __name__ == "__main__":
    public_url = ngrok.connect(5000)
    # Obtener la URL pública generada por Ngrok
    public_url_str = str(public_url)
    print('URL pública de Ngrok:', public_url_str)
    app.run()


'''
if __name__ == "__main__":
    app.run('0.0.0.0',port=5000)
'''
```

```
URL pública de Ngrok: NgrokTunnel: "https://b984-38-25-22-239.ngrok-free.app" -> "http://localhost:5000"
[33m * Tip: There are .env files present. Install python-dotenv to use them.[0m
 * Serving Flask app 'app'
 * Debug mode: off
[31m[1mWARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.[0m
 * Running on http://127.0.0.1:5000
[33mPress CTRL+C to quit[0m
```

Entrar a la url proporcionado por Ngrok o localmente por http://127.0.0.1:5000/

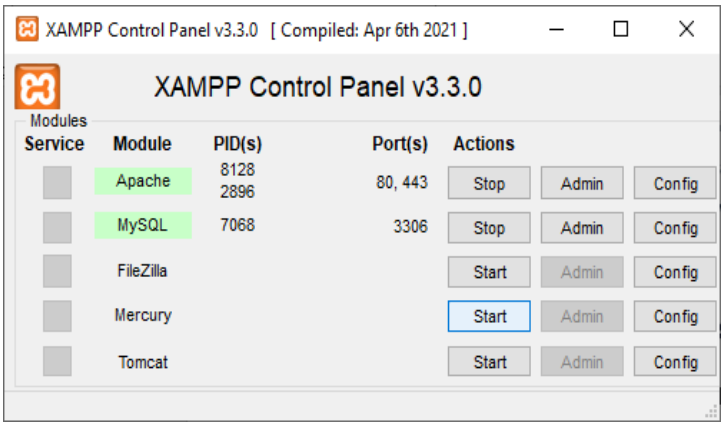


Simulación Dinámica en Sistema Operativo móvil

Introducción. – Las autoridades con frecuencia están fuera de sus oficinas ,por lo tanto, el presente proyecto les permitirá observar el comportamiento del proceso del sistema transporte en modo real desde distintos dispositivos entre los cuales se encuentran, el teléfono celular, la Tablet, la laptop o monitor de escritorio. De tal

manera que estén al tanto de la vida diaria de los procesos en transporte y pueden tomar decisiones más oportunas en forma remota.

Para conocer la información en modo real, se ha diseñado el modelo dinámico en software vensim y bajo sistema operativo Windows 10, usando nuestro sistema operativo como un servidor local por medio de la herramienta Xampp el cual que contendrá el archivo Vensim, la cual por medio de un desarrollo en Flask para la plataforma web, con distintos mecanismos de desarrollo se tendrá acceso al archivo vensim alojado en nuestro servidor local y por medio de la integración con la herramienta ngrok la cual es usada en forma gratuita, nos permitirá conectar o exponer nuestro **desarrollo web localmente** en forma **pública.** Y así tener nuestra aplicación local para la comunidad mundial o pueda ver la simulación sin la necesidad de subir la aplicación desarrollada a un servidor de paga. Para este uso como medias previas se debe activar el servidor XAMPP sus servicios de Apache y MySQL para la base de datos, debido que el desarrollo en Flask consume información de tablas de la BD la cuales interpretara los obtenido por Vensim sin tener la necesidad de modificar el software fuente.



Luego se hará uso de la conexión de base de datos usando **Python** para conocer el comportamiento del sistema por cada variable definida.

También decirles que se usa las librerías:

Ssl, flask, pyngrok, pysd, mysql-connector-python, python-decouple, numpy, urllib3, mpld3

NGROK, para usarlo tendremos un comando en la terminal a través del cual podremos exponer el puerto para ir a la a la web.

Para usar NGROK , considere los siguientes pasos:

1.- exponer nuestro entorno local para así no tener que depender de un servidor de producción.

2.-con flask, se crea la ruta de la web a publicar usando nuestro computador como servidor y la conexión de esta con el servidor local levantado con XAMPP

3.- la ruta creada automáticamente cuando diseño mi app.py, de esta manera nos brinda una url que se copia en la barra de direcciones de cualquier navegador utilizado entre los más comunes (Google Chrome, Microsoft Edge, etc)

3.- Nos conducirá a un site, donde se mostrará el proceso de simulación.