Reporte de la solución del problema.

Con realización de la prueba lo que se busca es:
- Crear una API Rest con Flask que implemente un sistema de inventario de productos con unas características mínimas.

Comprendiendo además unos requisitos mínimos:

Requisitos de Software

soft_req_001. Una entidad Tienda que almacene la información básica asociada a la misma (ie. nombre, dirección, etc.).
soft_req_002. Una entidad Producto que almacene la información básica asociada a la misma (ie. SKU, etc.).
soft_req_003. Poder agregar/asociar inventario a una tienda.
soft_req_004. Poder determinar si hay suficiente stock de un producto en la tienda.
soft_req_005. Interactuar con el sistema de inventario de forma asíncrona.
soft_req_006. Improvisar cualquier información que no haya sido suministrada o que se considere.

Requisitos de Hardware

hard_req_001. Uso de herramienta para pruebas de peticiones (imsomnia).

Requisitos de ambiente o de sistema

sys_req_001. No usar frameworks (ie. Django, etc.), con la excepción de Flask, Flask-RESTX, Psycopg2 etc. lo demás debe ser lo mas “vanilla” posible.
sys_req_002. Hacer uso de una BD Postgres.

Planteamiento de la solución del problema.

Para empezar a trabajar primeramente se debe crear un ambiente virtual que permita crear todas las dependencias a utilizar en la realización del código que va a plasmar la solución del problema.

Primera parte: Configurar un entorno virtual con virtualenv para trabajar con las librerías necesarias. Además poder configurar con el uso del servidor Flask diferentes entornos (desarrollo, pruebas y producción) para todo lo relacionado al funcionamiento de la aplicación.
Segunda parte: Configurar una base de datos PostgreSQL junto con SQLAlchemy y Alembic para manejar las migraciones y las actualizaciones a la base de datos.
Tercera parte: Agregar la lógica de back-end donde se realicen las peticiones (GET, POST, PUT) y respuestas en formato JSON que la aplicación estará manejando sobre la tienda, los productos y su inventario. Utilizar SQLAlchemy para mapear la relación de objetos en flask, o bien, dicho de otra forma se emplea para serializar los objetos y tener las relaciones entre las tablas de la base de datos y los objetos de los modelos.
Cuarta parte: Implementar bucles de eventos con threads (hilos) para la ejecución de corrutinas y de esa forma poder soportar peticiones asíncronas, para ello se podría emplear asyncio.
Quinta parte: Comprobar mediante pruebas unitarias de interacción con el usuario el funcionamiento de las diferentes funcionalidades.

Requisitos para funcionamiento de la aplicación

Ambiente de desarrollo:
- Sistema Operativo: Ubuntu 18.04.5 LTS
- memoria: 15.5 GiB
- ProcesadorIntel Core i5-8250U CPU @1.60GHzx8
- Graficos: Intel UHD Graphics 620 (KBL GT2)
- GNOME 3.2.8.2
- Tipo SO: 64 bits
- Disco 89.1 GB

Lenguaje de desarrollo de Software
- Python 3.6.9

Instalación de módulos en el entorno virtual de python
alembic==1.6.5
anyio==3.3.0
asgiref==3.4.1
async-generator==1.10
bleach==4.0.0
certifi==2021.5.30
charset-normalizer==2.0.4
click==7.1.2
contextvars==2.4
dataclasses==0.8
docutils==0.17.1
Flask==1.0.3
flask-app==1.0.1
flask-marshmallow==0.14.0
Flask-Migrate==2.5.2
Flask-Script==1.0.0
Flask-SQLAlchemy==2.4.0
Flask-Testing==0.7.1
flask-unittest==0.1.2
greenlet==1.1.1
h11==0.12.0
httpcore==0.13.6
httpx==0.19.0
idna==3.2
immutables==0.16
importlib-metadata==4.6.4
itsdangerous==2.0.1
Jinja2==3.0.1
lxml==4.6.3
Mako==1.1.5
Markdown==3.3.4
MarkupSafe==2.0.1
marshmallow==3.13.0

marshmallow-sqlalchemy==0.26.1
packaging==21.0
pkg-resources==0.0.0
pkginfo==1.7.1
psycopg2-binary==2.8.3
Pygments==2.10.0
pyparsing==2.4.7
python-dateutil==2.8.2
python-dotenv==0.10.3
python-editor==1.0.4
readme-renderer==29.0
requests==2.26.0
requests-toolbelt==0.9.1
rfc3986==1.5.0
six==1.16.0
sniffio==1.2.0
SQLAlchemy==1.4.23
tqdm==4.62.2
twine==1.13.0
typing-extensions==3.10.0.0
urllib3==1.26.6
webencodings==0.5.1
Werkzeug==2.0.1
zipp==3.5.0

Visualización de la estructura del modelo de la base de datos

En la Imagen 1 se muestra la relación entre los modelos Stores, Products y StoreProducts de la base de datos (el workbench se realizó en mysql pero su implementación fue realizada en Postgres)

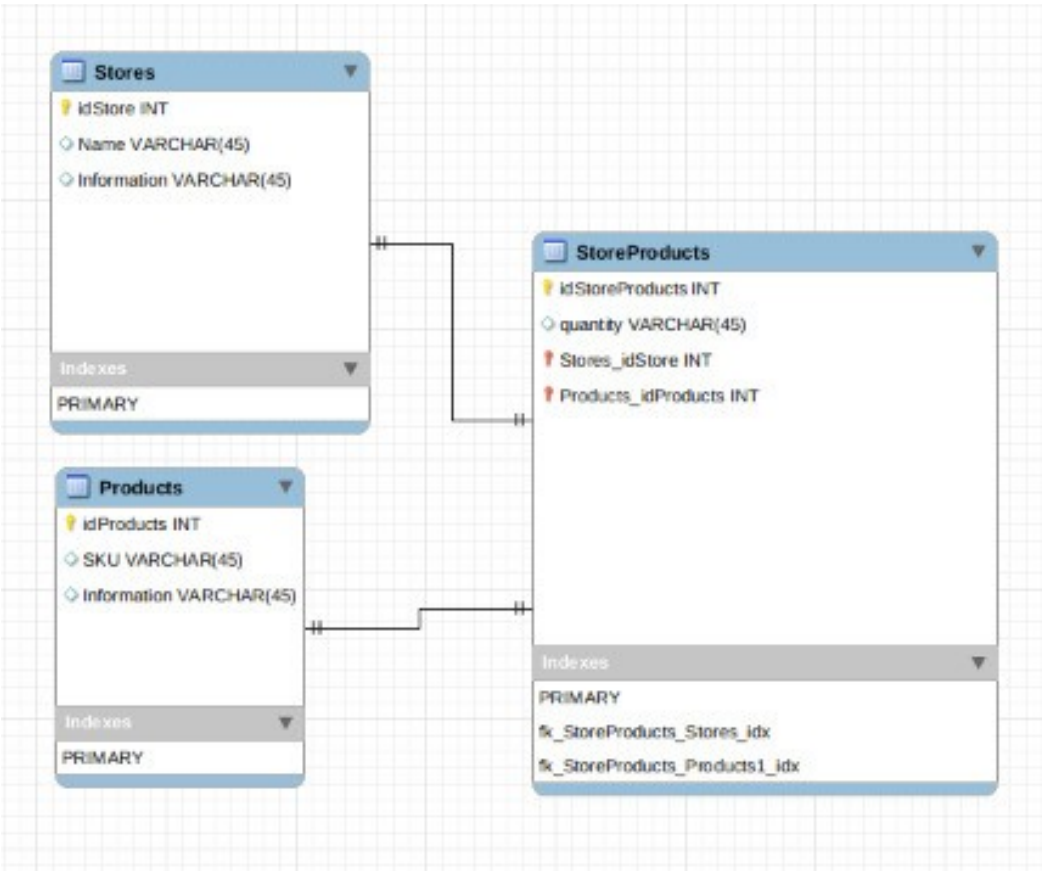

Imagen 1: Workbench base de datos de los modelos creados.

En la Imagen 2 y la Imagen 3 se muestra haciendo uso del programa pgAdmin III, donde refleja las características de la base de datos:
- El nombre de la base de datos en Postgres a conectarse “Shop_products”.
- Las tablas creadas stores, products, y storeProducts y sus campos.

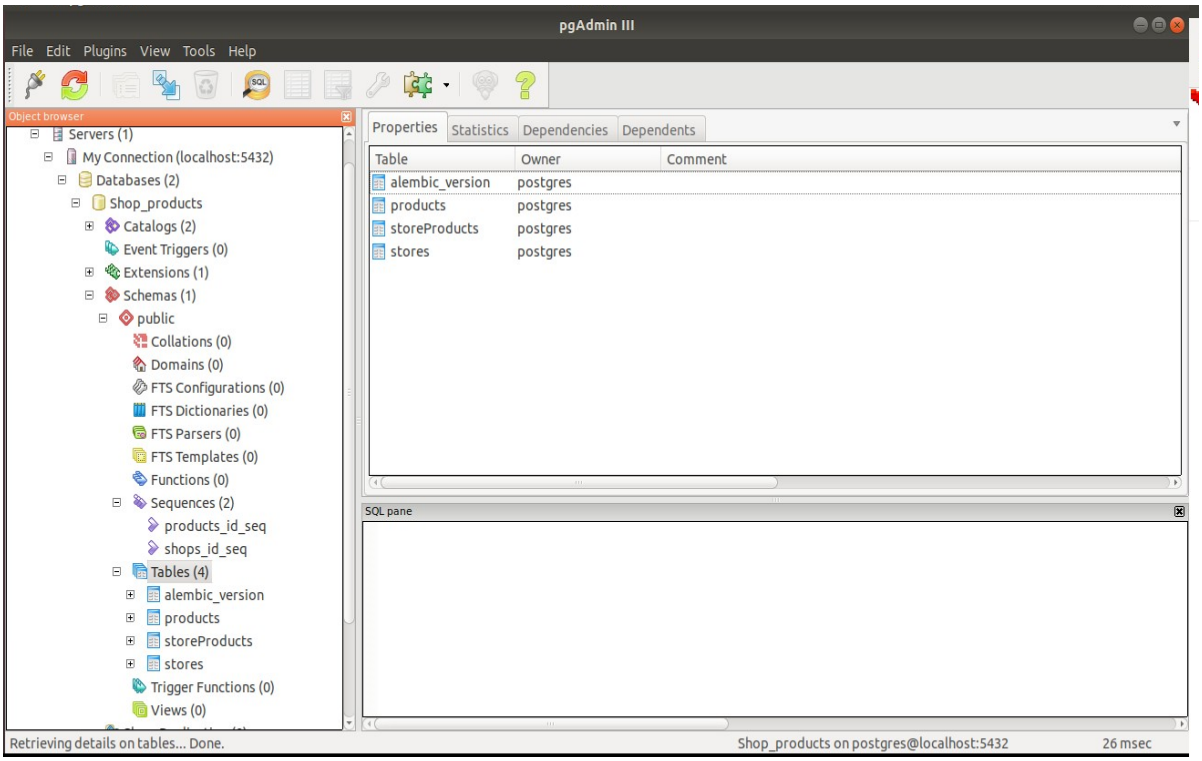

Imagen 2: Base de datos “Shop_products” creada en Postgres.

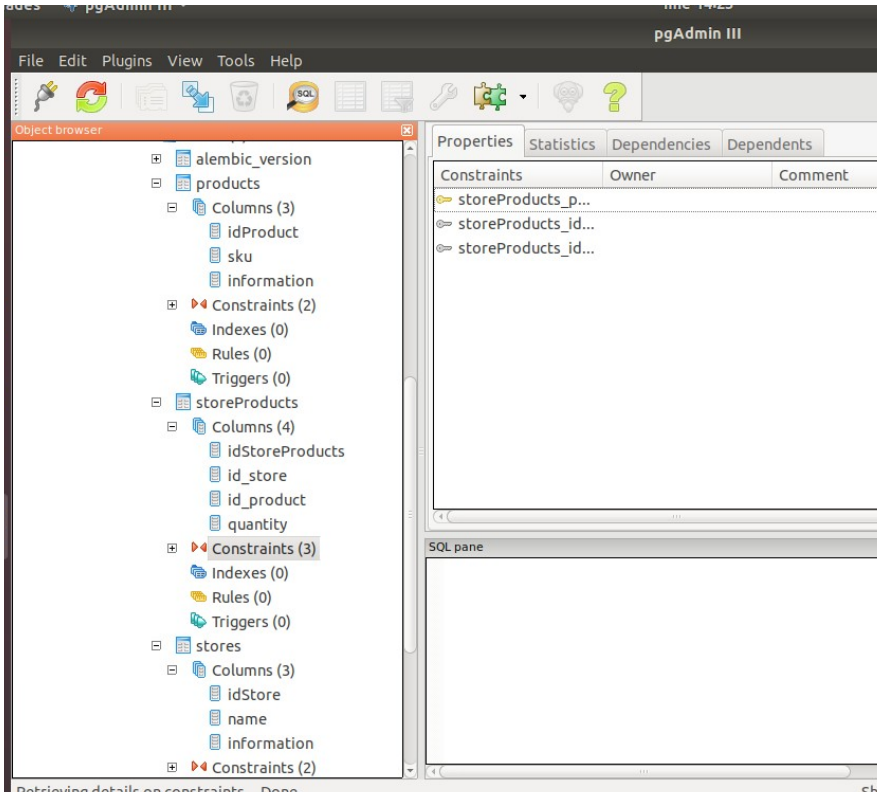

Imagen 3: Campos de las tablas de stores, products y storeProduct de la base de datos “Shop_products” creada en Postgres.

Estructura del código

```
Products_RestAPI
├── app
│   ├── __pycache__
│   ├── app.py
│   ├── db.py
│   ├── models.py
│   └── test
│       ├── __pycache__
│       └── testy.py
├── config
│   ├── __pycache__
│   └── config.py
├── docs
│   ├── installation.txt
│   ├── Report.odt
│   └── requirements.txt
├── installation.txt
├── logs
│   └── app.log
├── manage.py
├── migrations
│   ├── alembic.ini
│   ├── env.py
│   ├── __pycache__
│   ├── README
│   ├── script.py.mako
│   └── versions
│       ├── b39a080100ef_.py
│       └── __pycache__
│           └── b39a080100ef_.cpython-36.pyc
└── README.txt
```

Comandos de ejecución de API (Products_RestAPI**)**

Para crear las tablas y dar uso a los modelos de la base generada se deben ejecutar los siguientes comandos: El primer comando sirve para crear una estructura de directorio y ficheros necesarios para la ejecución de esta extensión. Se ejecuta por una sólo vez y al principio.

-$ **python3 manage.py db init**

El segundo comando se utiliza para revisar actualizaciones en los modelos y ayuda a generar los ficheros de migración de base de datos con los cambios detectados.

-$ **python3 manage.py db migrate**

El tercer comando se utiliza para subir o hacer la migración los datos a la base de datos.

-$ **python3 manage.py db upgrade**

Y por último, para la execución del servidor se tiene el cuarto comando:

-$ **python3 manage.py run**

Funcionamiento asíncrono de la aplicación

El funcionamiento principal de la aplicación radica en poder hacer peticiones al servidor sin tener que esperar que se procese una y luego la otra, o bien sea de forma asíncrona. Para ello es importante considerar la Imagen 4, donde se observa una aplicación con manejo síncrono y otro asíncrono, para lo cual deja ver claramente la secuencialidad de los sistemas síncronos, mientras resalta la optimización de tiempo de los sistemas asíncronos.

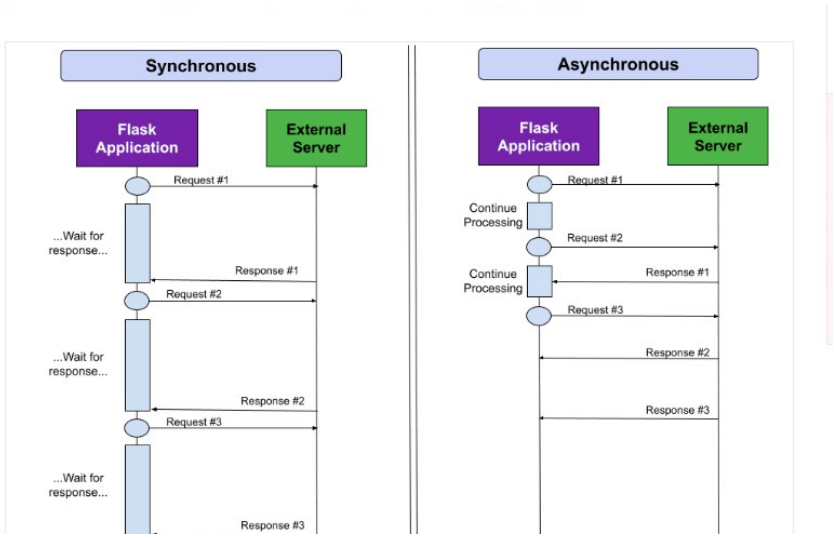

Imagen 4: Comparación de peticiones síncronas vs peticiones asíncronas.

El uso por de la librería Flask en por lo menos la versión 2.0, y la librería asyncio permite crear corrutinas para manejo de diferentes peticiones al tiempo. En la Imagen 5 se muestra la ejecución de dos peticiones al mismo tiempo y bajo el esquema de threads en el que se estaría ejecutando la aplicación realizada.

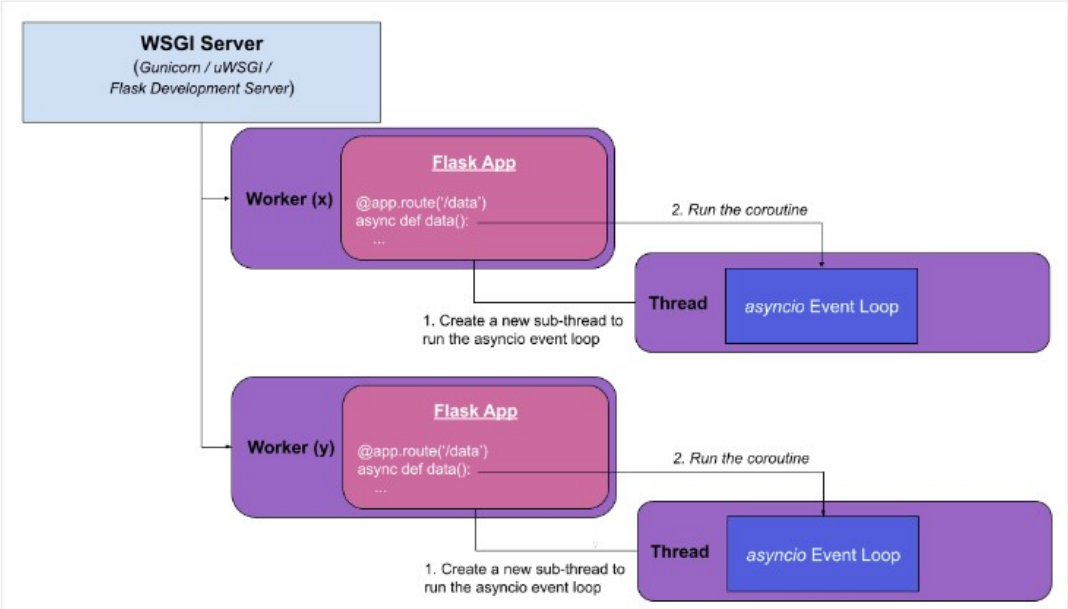

Imagen 5: Ejecución de subrutinas a través de threads (hilos) eempleando la librería de python asyncio.

Actividades pendientes por hacer

- Generar archivo de pruebas y cumplimiento de requisitos.
- Generar pruebas automatizadas con Unittest para verificación de funcionamiento de código.
- Crear pruebas de caja negra
- Crear pruebas de caja blanca
- Crear pruebas de caja gris