Simulador de Crecimiento de Cultivos – Yeremy Vega Pineda 20242020233

1. Descripción general del proyecto

El sistema es un microservicio web desarrollado con **Flask**, cuyo objetivo es estimar un **puntaje de crecimiento** de una parcela a partir de dos variables básicas: **horas de luz** y **nivel de riego**.

El usuario puede interactuar con el sistema desde dos frentes:

1. **Un formulario HTML**, donde ingresa manualmente los parámetros.
2. **Una API REST en JSON**, que permite enviar datos desde clientes externos y recibir las predicciones programáticamente.

Cada simulación se **registra automáticamente** en una base de datos SQLite mediante SQLAlchemy, quedando almacenados los parámetros de entrada, el puntaje generado y la fecha de la simulación.

El propósito del proyecto es extender la app mínima desarrollada en la Guía 21 hacia una aplicación más organizada, modular y cercana a un microservicio real, en este caso una aplicacion web de simular el crecimiento de cultivos con dos entradas.

3. Contexto y justificación técnica

Se solicito un proyecto que integre persistencia estructurada, servicios HTTP, validaciones disciplinadas y uso de un modelo entrenado. La solución implementada cumple cada requisito:

- ***Modelo entrenado propio:*** El proyecto incluye un script (files_for_training_model/train.py) que genera un modelo sencillo con scikit-learn, almacenado luego en models/model.pkl. Esto garantiza reproducibilidad y evita depender de modelos “mágicos”.
- ***Persistencia con SQLAlchemy:***
  Se eligió SQLAlchemy porque:
  - ★ Reduce el uso de SQL manual.
  - ★ Permite representar la tabla Simulacion como una clase Python.

    - ★ Facilita migrar de SQLite a otro motor si el proyecto crece.

- ***Lógica validada y centralizada:*** La validación no está en app.py sino en services/validacion.py. Esto evita duplicar lógica entre HTML y API, y mejora la mantenibilidad.

- ***Diseño modular:*** Separar app.py, services/, database.py y models/ permite expandir el proyecto fácilmente hacia:
    - ★ sensores por sockets
    - ★ tareas concurrentes
    - ★ Dashboards
    - ★ endpoints adicionales

Cumple la idea principal del proyecto final: mostrar una arquitectura clara y preparada para crecer.

4. Arquitectura técnica

La arquitectura se divide en cuatro capas:

1) ***Capa de presentación – Flask (app.py)***

Define:

+ Ruta HTML principal (GET /)
+ Ruta para procesar el formulario (POST /predict)
+ Endpoint REST JSON (POST /api/predict)

app.py no realiza cálculos ni validaciones directamente; solo coordina.

2) Capa de servicios – services/

+ ***validacion.py***
    - Revisa tipos, rangos y estructura de datos.
    - Garantiza que el sistema nunca procese datos incompletos o inválidos.
+ ***simulaciones.py***
    - Carga el modelo entrenado solo una vez (lazy loading).
    - Construye el vector de características.
    - Ejecuta la predicción.
    - Registra la simulación en la base de datos mediante SQLAlchemy.

3) Capa de persistencia – database.py

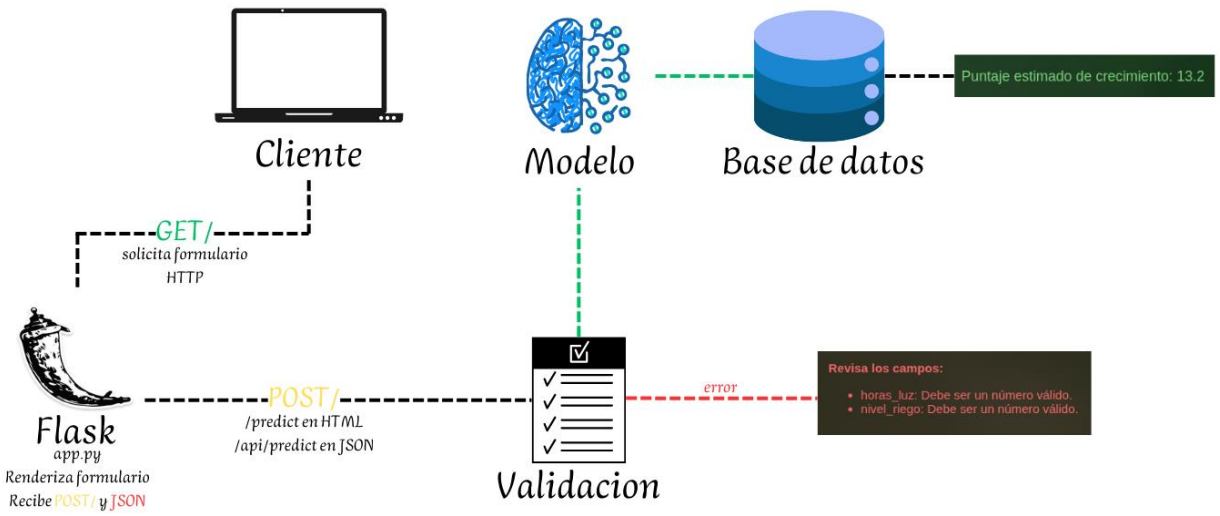
Contiene:

- Configuración del engine y la sesión.
- La clase ORM Simulacion.
- El context manager sesion_bd() que controla commit / rollback.
- La tabla incluye campos como:
- id
- horas_luz
- nivel_riego
- puntaje_crecimiento
- creado_en

4) Capa de modelo entrenado – models/model.pkl

Generado con scikit-learn.

Se utiliza desde simulaciones.py para obtener predicciones.



5. Flujo completo de datos

1. Solicitud inicial del cliente (GET /)

El cliente ya sea un navegador o un consumidor HTTP envía una petición GET / al servidor Flask.

Flask responde renderizando el formulario HTML donde el usuario puede ingresar las variables necesarias para la simulación.

2. Envío de datos al servidor (POST /predict o POST /api/predict)

Cuando el usuario ingresa las horas de luz y el nivel de riego:

- ★ Si usa el formulario HTML → se envía un POST /predict.
- ★ Si usa un cliente externo (Postman, script, etc.) → se envía un POST /api/predict con un cuerpo JSON.

En ambos casos, Flask recibe los datos y los envía al módulo de validación.

3. Validación de los datos (services/validacion.py)

El módulo de validación:

- ★ Verifica que los campos estén presentes.
- ★ Asegura que los valores sean numéricos.
- ★ Comprueba que estén dentro del rango permitido.

Si algo falla, se responde inmediatamente:

- × Con un mensaje de error en HTML (formulario).
- × Con un JSON de error (API).

Si los datos son correctos, continúan hacia la etapa de simulación.

4. Ejecución del modelo de predicción (services/simulaciones.py)

Con los datos validados:

El módulo carga el modelo entrenado (model.pkl) si aún no está en memoria y:

- ★ Construye el vector de características con las variables ingresadas.
- ★ Llama al método de predicción del modelo.
- ★ Obtiene el puntaje estimado de crecimiento.
- ★ Este puntaje se devuelve junto con los valores utilizados.

5. Persistencia de la simulación (SQLAlchemy + SQLite)

El resultado de la simulación se registra en la base de datos:

- ★ Se crea una instancia de Simulacion.
- ★ Se guardan las horas de luz, el nivel de riego, la predicción y la fecha.
- ★ La transacción se confirma para que quede almacenada de forma permanente.

Esto permite auditoría y reutilización futura de los datos.

6. Respuesta final al cliente

Finalmente, el servidor responde según el origen de la solicitud:

- ★ HTML: muestra la página con el puntaje de crecimiento.
- ★ API JSON: devuelve un objeto estructurado con: los valores enviados, el puntaje generado y el id asignado en la base de datos.

6. Diseño e implementación

El diseño se planteó siguiendo una separación estricta de responsabilidades, de manera que cada parte del sistema cumple una función específica y fácilmente comprobable. La estructura del proyecto se organizó en cuatro capas: presentación, validación, simulación y persistencia.

- ***app.py (capa de presentación):***
  Contiene únicamente las rutas HTTP. GET / muestra el formulario HTML y POST /predict / POST /api/predict procesan las solicitudes. No implementa validaciones ni cálculos; solo coordina el flujo. Esto reduce el acoplamiento y facilita la lectura del código.
- ***services/validacion.py:***
  Aquí se centralizan todas las verificaciones de entrada. Se revisa la presencia de los campos, su tipo numérico y el rango permitido. Se optó por devolver diccionarios con mensajes específicos en lugar de errores genéricos, para mantener un control detallado de las fallas.
- ***services/simulaciones.py:***
  Implementa la lógica de negocio. Carga el modelo entrenado (.pkl) solo una vez para evitar sobrecostos de E/S, construye el vector de entrada y ejecuta la predicción. También prepara la estructura final que se enviará a la base de datos.
- ***database.py:***
  Encapsula la configuración del ORM SQLAlchemy. Define la clase Simulacion con columnas claras y un context manager (sesion_bd) que garantiza commits y rollbacks limpios. Esta decisión evita fugas de sesión y facilita que la persistencia sea atómica.
- ***Manejo de errores:***
  Se evitó lanzar excepciones hacia Flask. En su lugar, las funciones de validación devuelven estructuras bien definidas indicando éxito o error. Esto permite un control fino desde las rutas y evita páginas de error no deseadas.

En conjunto, el diseño prioriza claridad, extensibilidad y facilidad de pruebas, manteniendo el código simple y organizado.

7. Integración de técnicas del curso

El proyecto integra de forma concreta los elementos vistos durante el semestre:

- **Persistencia estructurada:**
  SQLAlchemy gestiona la base SQLite creando un modelo ORM que permite registrar cada simulación con sus parámetros y resultados.
- **Servicios HTTP:**
  Flask implementa rutas HTML y una API REST en JSON. La API sigue buenas prácticas: uso de POST para procesamiento, códigos 200/400 según el resultado y serialización consistente.
- **Serialización y JSON:**
  Las respuestas API devuelven objetos JSON estructurados, facilitando su consumo por clientes externos o pruebas automatizadas.
- **Modelo entrenado (Machine Learning):**
  Aunque no se vio explicitamente en el curso, se aplico Machine Learning. Hay un sistema que integra un modelo real almacenado en models/model.pkl, entrenado con datos sintéticos mediante scikit-learn. Esto cumple el requisito de usar un artefacto externo procesado previamente.
- **Diseño modular:**
  La app está dividida en módulos independientes, permitiendo que en un futuro se pueda agregar concurrencia con threads, sockets, colas o sensores sin reescribir la arquitectura.
- **Preparación para concurrencia:**
  Aunque no se implementaron hilos explícitos, el código evita variables globales mutables (excepto el modelo cargado en solo-lectura), lo cual es compatible con threads o servidores WSGI con múltiples workers.

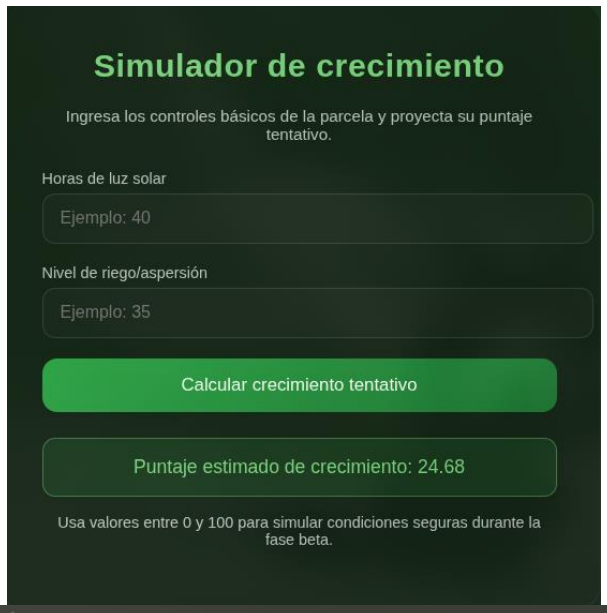
8. Pruebas y validación

Se realizaron pruebas manuales y por consola para verificar el funcionamiento de la aplicación:

Escenarios evaluados

Entrada válida por HTML:

El formulario devuelve correctamente el puntaje, mostrando los valores ingresados y permitiendo nuevas simulaciones.



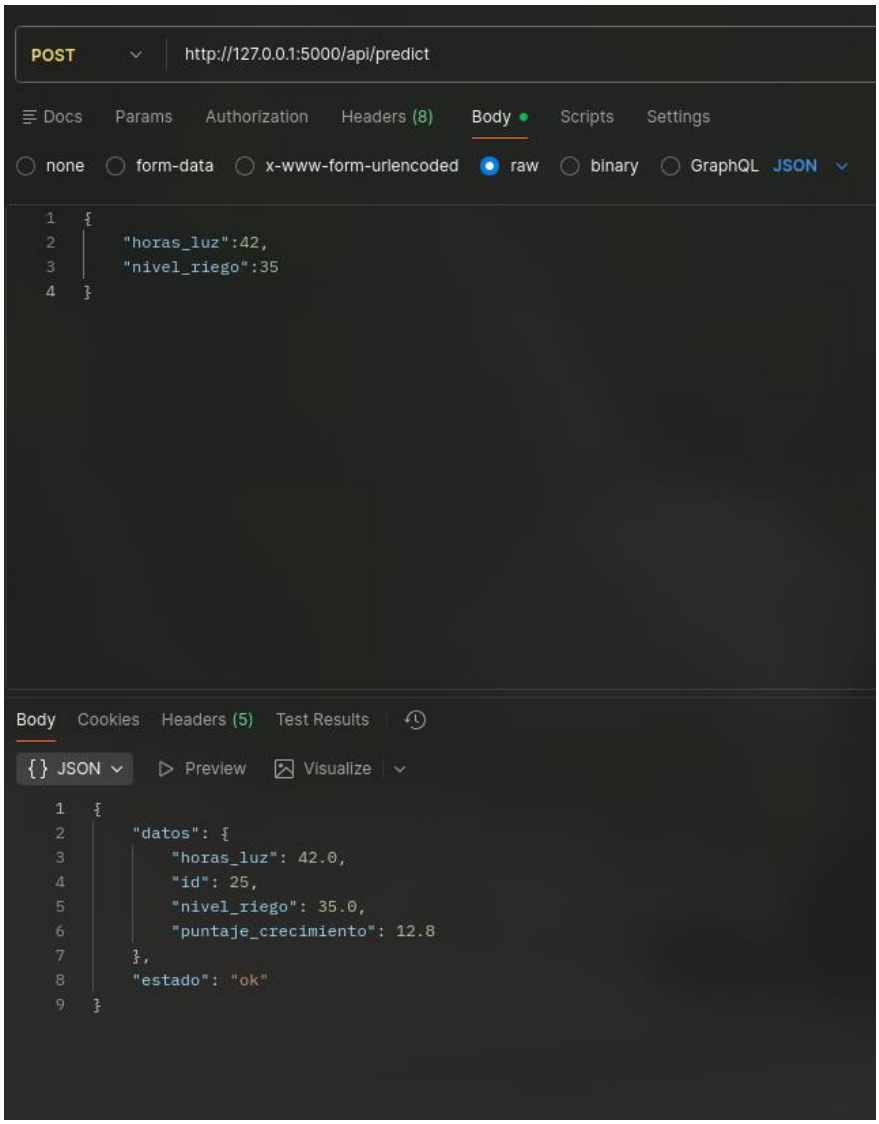

```
  warnings.warn(
127.0.0.1 - - [03/Dec/2025 01:36:15] "POST /predict HTTP/1.1" 200 -
```

Entrada válida por API:

Usando Postman, se validó que el sistema procesara solicitudes JSON como:

{"horas_luz": 5, "nivel_riego": 40}

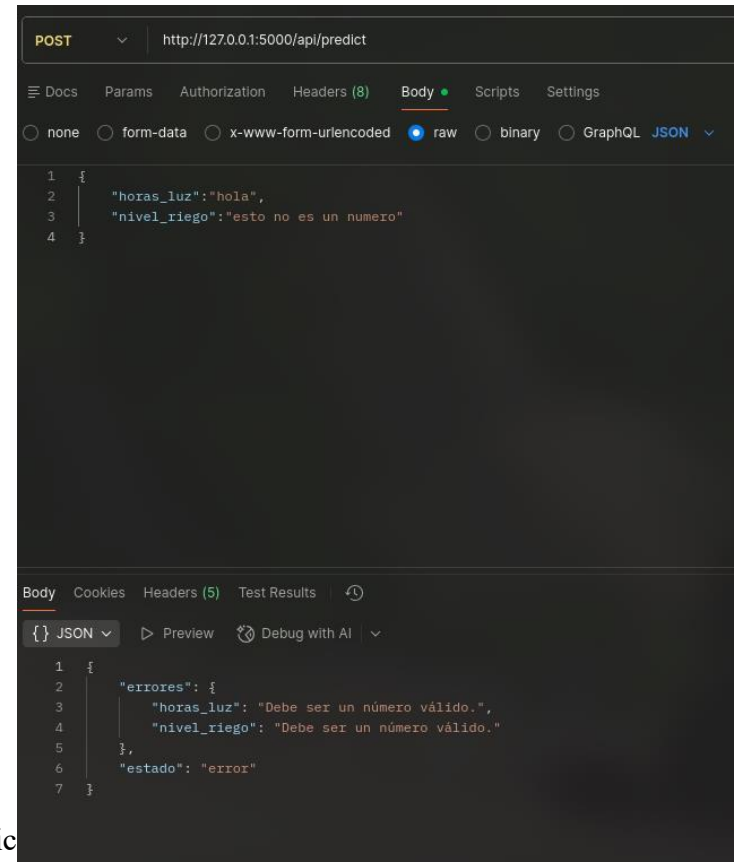
devolviendo ID de simulación y puntaje.



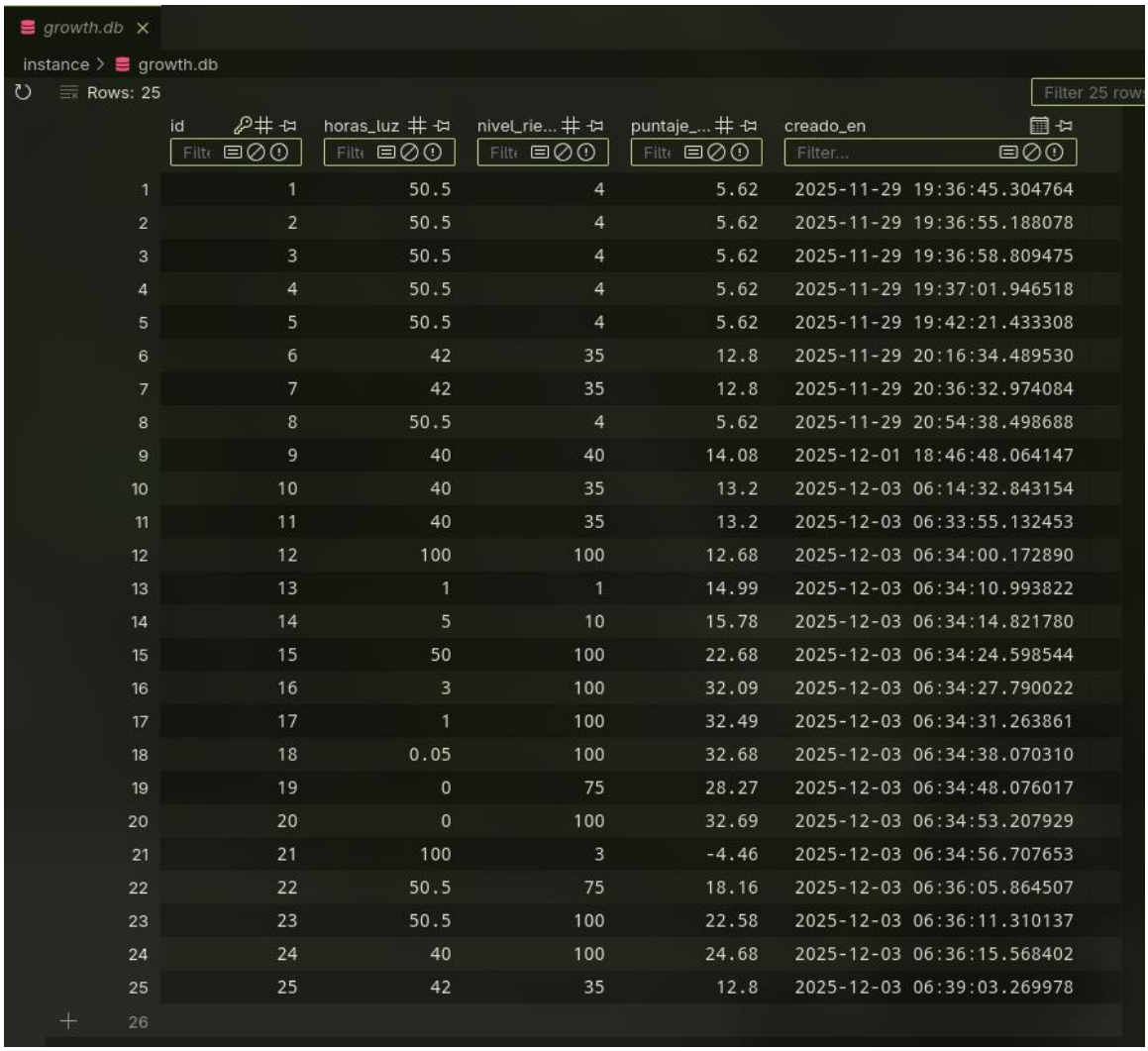
Errores de validación:

Se probaron faltas de campos, valores no numéricos y valores fuera de rango. El sistema siempre respondió con mensajes específicos, tanto en HTML como en JSON.

```
127.0.0.1 - - [03/Dec/2025 01:40:10] "POST /api/predict HTTP/1.1" 400 -
127.0.0.1 - - [03/Dec/2025 01:40:34] "POST /predict HTTP/1.1" 200 -
```



Se revisó directamente la base de datos, verific
nueva fila con fecha, parámetros y resultado.


growth.db

instance > growth.db

Rows: 25

Filter 25 row

	id	horas_luz	nivel_rie...	puntaje_...	creado_en
	Filte	Filte	Filte	Filte	Filter...
1	1	50.5	4	5.62	2025-11-29 19:36:45.304764
2	2	50.5	4	5.62	2025-11-29 19:36:55.188078
3	3	50.5	4	5.62	2025-11-29 19:36:58.809475
4	4	50.5	4	5.62	2025-11-29 19:37:01.946518
5	5	50.5	4	5.62	2025-11-29 19:42:21.433308
6	6	42	35	12.8	2025-11-29 20:16:34.489530
7	7	42	35	12.8	2025-11-29 20:36:32.974084
8	8	50.5	4	5.62	2025-11-29 20:54:38.498688
9	9	40	40	14.08	2025-12-01 18:46:48.064147
10	10	40	35	13.2	2025-12-03 06:14:32.843154
11	11	40	35	13.2	2025-12-03 06:33:55.132453
12	12	100	100	12.68	2025-12-03 06:34:00.172890
13	13	1	1	14.99	2025-12-03 06:34:10.993822
14	14	5	10	15.78	2025-12-03 06:34:14.821780
15	15	50	100	22.68	2025-12-03 06:34:24.598544
16	16	3	100	32.09	2025-12-03 06:34:27.790022
17	17	1	100	32.49	2025-12-03 06:34:31.263861
18	18	0.05	100	32.68	2025-12-03 06:34:38.070310
19	19	0	75	28.27	2025-12-03 06:34:48.076017
20	20	0	100	32.69	2025-12-03 06:34:53.207929
21	21	100	3	-4.46	2025-12-03 06:34:56.707653
22	22	50.5	75	18.16	2025-12-03 06:36:05.864507
23	23	50.5	100	22.58	2025-12-03 06:36:11.310137
24	24	40	100	24.68	2025-12-03 06:36:15.568402
25	25	42	35	12.8	2025-12-03 06:39:03.269978
26					

Robustez del modelo:

Se probó un amplio rango de valores para confirmar que el modelo respondiera sin errores y generara resultados coherentes.

Resultados

El sistema operó correctamente en todas las pruebas, procesando entradas válidas y controlando adecuadamente cada error. La base de datos registró todas las simulaciones, y la API REST mostró un comportamiento estable y consistente.

9. Discusión

El sistema cumplió los objetivos planteados: proporcionar un flujo claro desde la entrada del usuario hasta la persistencia del resultado, integrando un modelo entrenado real y manteniendo un diseño modular.

Fortalezas observadas

- ✓ La separación de responsabilidades hizo que la depuración fuera muy sencilla.
- ✓ La API REST surgió casi sin esfuerzo adicional, gracias a la modularidad.
- ✓ El uso de SQLAlchemy permitió observabilidad real sobre los datos producidos.

Dificultades encontradas

- × Integrar validaciones estrictas sin duplicar código en HTML y API tomó tiempo.
- × Reentrenar el modelo requería un entorno separado para evitar inflar los requirements del runtime. (En ese momento no tuve encuenta que el proyecto original era relativamente antiguo, por eso opte por actualizarlo cosa que podia evitar simplemente entrando al entorno del repositorio del cual se inspiro el proyecto)
- × En etapas iniciales, era fácil generar errores silenciosos si el formulario enviaba strings vacíos o caracteres no numéricos.

Valoración general

El sistema se comporta de forma estable y predecible. Aunque el modelo es simple y usa datos sintéticos, la estructura del proyecto soporta adecuadamente escenarios reales. El flujo entre cliente → servidor → modelo → base de datos demostró ser sólido.

10. Conclusiones

- El proyecto logró implementar una arquitectura modular y extensible, adecuada para integrar nuevas funciones sin alterar la base del sistema.

- Se puso en práctica persistencia, servicios HTTP, validaciones, uso de modelos entrenados y buenas prácticas vistas durante el semestre.

- El trabajo permitió comprender de forma práctica cómo exponer modelos de Machine Learning mediante una API real.

- La claridad del diseño facilita que el proyecto pueda ampliarse hacia sensores, dashboards o concurrencia.

11. Limitaciones del sistema

- El modelo utiliza datos sintéticos, por lo que no representa comportamientos agrícolas reales.
- No existe autenticación ni control de usuarios.
- La interfaz HTML es funcional, pero minimalista.
- No hay pruebas automatizadas; todas las pruebas fueron manuales.
- No se implementó concurrencia explícita, aunque el sistema está preparado para ella.

12. Trabajo futuro

- Conectar sensores simulados mediante sockets que alimenten el sistema en tiempo real.
- Añadir un módulo concurrente que procese simulaciones en background usando ThreadPoolExecutor.
- Implementar un dashboard gráfico en HTML para visualizar tendencias del crecimiento.
- Reentrenar el modelo con datos reales recolectados en laboratorio.
- Añadir autenticación y control de acceso para la API.
- Migrar a PostgreSQL si el número de simulaciones crece significativamente.

Subraye las que mas me llaman la atencion, pero en general este proyecto podria tomar muchos caminos a futuro

13. Guía mínima de ejecución

1) python -m venv .venv

2.1) .\nombre_del_entorno\Scripts\activate (en Windows)

2.2) source .venv/bin/activate (en Linux/macOS)

3) pip install -r requirements.txt

4) python app.py

5) En un navegador colocar la URL *http://localhost:5000/*

14. Referencias

Flask
Pallets. (2024). *Flask documentation* (versión 3.x). https://flask.palletsprojects.com/

SQLAlchemy
Bayer, M. (2024). *SQLAlchemy 2.x documentation*. https://docs.sqlalchemy.org/

scikit-learn
Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., … Duchesnay, É. (2011). *Scikit-learn: Machine learning in Python*. Journal of Machine Learning Research, 12, 2825–2830. https://scikit-learn.org/

SQLite (motor de base de datos)
SQLite Project. (2024). *SQLite documentation*. https://www.sqlite.org/docs.html

SQLite Viewer (extensión o herramienta)
SQLite Viewer. (2024). *SQLite Viewer – Extension for Visual Studio Code*. https://marketplace.visualstudio.com/items?itemName=qwtel.sqlite-viewer

Guía 21 del profesor
Torres Castillo, J.R. (2024). *Guía 21 – Programación Avanzada* Programación Web – App Web con Flask para Predicción de Enfermedades Cardíacas. Facultad de Ingeniería, Universidad Distrital Francisco José de Caldas.

Video de DigitalSreeni
DigitalSreeni. (2022). *How to deploy your trained machine learning model into a local web application* [Video]. YouTube. https://youtu.be/bluclMxiUkA?t=1342

Repositorio clonado (solo directorio 268)
Sreeni, B. (2024). *python_for_microscopists* [Repositorio GitHub]. GitHub. https://github.com/bnsreenu/python_for_microscopists

15. Enlace al video explicativo

https://youtu.be/demo-simulador-cultivo