

TRABAJO INTEGRADOR

Braulio Silva

Programación IV

Pontificia Universidad Católica del Ecuador

28 de enero de 2026

Resumen Ejecutivo

El presente proyecto detalla el desarrollo e implementación de una aplicación web orientada al cálculo de rutas óptimas entre ciudades del Ecuador. El sistema utiliza una arquitectura basada en micro-frameworks (Flask), integración con bases de datos relacionales (MySQL) y algoritmos de teoría de grafos (Dijkstra) para resolver problemas de conectividad en tiempo real. Se destaca la implementación de una interfaz gráfica moderna, consumo de APIs de geolocalización y paneles administrativos de auditoría.

Índice

1. Configuración del Proyecto y Estructura Modular	3
1.1. Arquitectura de Software	3
2. Conexión e Integración con MySQL	3
2.1. Modelado de Datos	3
3. Implementación CRUD	4
4. Gestión del Grafo y Algoritmo de Rutas	4
4.1. Lógica de Conectividad Híbrida	4
4.2. Implementación de Dijkstra	4
5. Interfaz de Usuario (UI/UX)	5
6. Integración de APIs y Servicios de Geolocalización	5
6.1. API de OpenStreetMap (OSM)	5
6.2. API de OSRM (Open Source Routing Machine)	5
7. Rutas y Controladores en Flask	5
8. Calidad del Código y Buenas Prácticas	6
9. Funcionalidades Extra y Valor Agregado	6
10. Conclusiones	6

1. Configuración del Proyecto y Estructura Modular

1.1. Arquitectura de Software

El sistema ha sido construido sobre el lenguaje **Python**, utilizando **Flask** como núcleo del servidor web. Para garantizar la escalabilidad y mantenibilidad del código, se adoptó un patrón de diseño basado en **Modelo-Vista-Controlador (MVC)**.

La organización del proyecto evita la estructura monolítica mediante el uso de **Blueprints**, lo que permite desacoplar la lógica en módulos independientes:

- **Módulo de Autenticación (auth_routes):** Gestiona sesiones, login seguro y protección de rutas.
- **Módulo Principal (main_routes):** Contiene el motor algorítmico, la gestión del grafo y la renderización del mapa.
- **Módulo Administrativo (admin_routes):** Controla la auditoría, reportes y visualización de logs del sistema.

2. Conexión e Integración con MySQL

La persistencia de datos se maneja a través de un motor de base de datos relacional **MySQL**, elegido por su robustez y velocidad en transacciones de lectura. La integración con Flask se realiza mediante **SQLAlchemy**, un ORM (Object-Relational Mapper) que abstrae las consultas SQL crudas en objetos de Python.

2.1. Modelado de Datos

El esquema relacional fue diseñado para mantener la integridad referencial y optimizar las consultas:

1. **Tabla Usuarios:** Almacena credenciales con contraseñas encriptadas (hashing) y define roles de acceso (Administrador/Usuario).
2. **Tabla Ciudades (Nodos):** Representa los vértices del grafo. Almacena metadatos geográficos críticos: Latitud y Longitud, necesarios para el cálculo de distancias.
3. **Tabla Historial (Logs):** Funciona como una bitácora de auditoría vinculada mediante llaves foráneas a la tabla de usuarios.

3. Implementación CRUD

El sistema implementa operaciones CRUD (Create, Read, Update, Delete) completas:

- **Creación (Create):** A través de formularios validados en el frontend, se permite la inserción de nuevas ciudades y el registro de usuarios.
- **Lectura (Read):** Se realizan consultas dinámicas para poblar el mapa con marcadores y listar los registros de actividad en el panel administrativo, utilizando paginación o límites de consulta.
- **Actualización Dinámica (Update Logic):** El sistema actualiza *automáticamente* la topología del grafo cada vez que se inserta un nuevo nodo, recalculando las vecindades.

4. Gestión del Grafo y Algoritmo de Rutas

Esta es la funcionalidad core del sistema. El grafo se construye **en tiempo de ejecución**, permitiendo que el mapa evolucione dinámicamente.

4.1. Lógica de Conectividad Híbrida

Para solucionar el problema de “nodos aislados”, se diseñó un algoritmo de conexión de doble criterio:

- **Criterio de Vecindad (K-Nearest Neighbors):** Cada ciudad se conecta obligatoriamente con sus **10 vecinos más cercanos**.
- **Criterio de Radio (Long Range):** Se establecen conexiones directas con cualquier ciudad dentro de un radio de **180 km**. Esto simula “autopistas regionales”.

4.2. Implementación de Dijkstra

Se programó el algoritmo de Dijkstra clásico optimizado con una **Cola de Prioridad (Priority Queue)**.

1. El algoritmo utiliza un *Min-Heap* para extraer siempre el nodo con menor costo acumulado.
2. Reconstruye el camino óptimo retrocediendo desde el destino hacia el origen.
3. El peso de las aristas es la distancia euclidiana, garantizando la ruta más corta geoméricamente.

5. Interfaz de Usuario (UI/UX)

La interfaz gráfica fue diseñada priorizando la experiencia de usuario (UX).

- **Diseño Visual (Cyberpunk/Neón):** Paleta de colores oscuros con acentos en cian y lila neón para facilitar la visualización sobre mapas satelitales.
- **Interactividad:** Uso de la librería **Leaflet.js** para renderizar mapas interactivos.
- **Responsividad:** Integración de **Bootstrap 5** para adaptación a dispositivos móviles.

6. Integración de APIs y Servicios de Geolocalización

El sistema no opera de manera aislada, sino que consume servicios externos mediante APIs para enriquecer la experiencia del usuario y validar los cálculos internos:

6.1. API de OpenStreetMap (OSM)

Se utiliza el servicio de teselas (Tiles) de **OpenStreetMap** como capa base del mapa. Esto permite visualizar carreteras, fronteras y geografía real sin costo de licenciamiento, garantizando un mapa base actualizado y detallado.

6.2. API de OSRM (Open Source Routing Machine)

Para proporcionar una comparativa técnica, el sistema integra la API de OSRM a través del plugin *Leaflet Routing Machine*.

- **Función:** Mientras que nuestro algoritmo interno (Dijkstra) calcula la ruta lineal óptima (Línea Violeta), OSRM calcula la ruta real por carretera (Línea Cian) considerando giros y sentido de las calles.
- **Consumo:** Se realizan peticiones asíncronas para obtener la geometría de la ruta y el tiempo estimado de viaje real, permitiendo al usuario contrastar la eficiencia teórica vs. práctica.

7. Rutas y Controladores en Flask

El backend actúa como una API RESTful simplificada, gestionando las peticiones HTTP:

- **Validación de Métodos:** Restricción de rutas (GET/POST).
- **Seguridad:** Uso de `@login_required` para proteger el panel administrativo.
- **Respuestas Feedback:** Implementación de *Flash Messages* para informar al usuario sobre el estado de las operaciones.

8. Calidad del Código y Buenas Prácticas

El desarrollo del software se adhirió a estándares de la industria:

- **Manejo de Excepciones:** Bloques `try-except` rodean las operaciones críticas.
- **Configuración Segura:** Variables sensibles separadas en `config.py`.
- **Modularidad:** Uso de paquetes Python para importaciones limpias.

9. Funcionalidades Extra y Valor Agregado

El proyecto supera los requisitos básicos mediante la integración de características avanzadas:

1. **Auditoría Administrativa:** Dashboard exclusivo para visualizar logs.
2. **Filtros Dinámicos:** Capacidad de filtrar historial por usuario y fecha.
3. **Sincronización Horaria:** Conversión automática de hora UTC a local mediante JavaScript.

10. Conclusiones

El sistema desarrollado representa una solución robusta y eficiente. La integración de algoritmos propios (Dijkstra) con APIs externas (OSRM/OSM) permite una validación cruzada de datos. La arquitectura modular en Flask y la interfaz moderna aseguran que el proyecto sea escalable, mantenible y profesional.