Universidad de San Carlos de Guatemala
Escuela de Ingeniería en Ciencias y Sistemas, Facultad de Ingeniería
Introducción a la programación y computación 2, 2do. Semestre 2025

IMPLEMENTACIÓN DE API REST CON PERSISTENCIA XML PARA FACTURACIÓN DE SERVICIOS EN LA NUBE

201907179 – Wilson Manuel Santos Ajcot

Resumen

El presente ensayo detalla el desarrollo de un sistema de facturación para servicios de infraestructura en la nube para la empresa ficticia Tecnologías Chapinas S.A. Se implementó una arquitectura de microservicios separando el backend (API REST con Flask) del frontend (interfaz web con Django). El backend gestiona la lógica de negocio, incluyendo la creación y manejo de recursos, categorías, configuraciones, clientes e instancias, aplicando el paradigma de Programación Orientada a Objetos (POO) para modelar las entidades. Se utiliza XML tanto para la carga inicial de datos como para la persistencia de la información en un archivo local, cumpliendo un requisito clave del proyecto. Se emplearon expresiones regulares para la validación de NITs y la extracción de fechas. El sistema culmina con la capacidad de procesar consumos y generar facturas detalladas por cliente. Se concluye que la arquitectura y tecnologías seleccionadas son viables para la gestión y facturación de los servicios descritos.

Palabras clave

API REST, Flask, Django, XML, Persistencia, Facturación, POO.

Abstract

Traducir (Traducción del Resumen al inglés) This essay details the development of a billing system for cloud infrastructure services for the fictitious company Tecnologías Chapinas S.A. A microservices architecture was implemented, separating the backend (REST API with Flask) from the frontend (web interface with Django). The backend manages the business logic, including the creation and handling of resources, categories, configurations, clients, and instances, applying the Object-Oriented Programming (OOP) paradigm to model the entities. XML is used both for the initial data loading and for the persistence of information in a local file, fulfilling a key project requirement. Regular expressions were used for NIT validation and date extraction. The system culminates in the ability to process consumption and generate detailed invoices per client. It is concluded that the selected architecture and technologies are viable for the management and billing of the described services.

Keywords

REST API, Flask, Django, XML, Persistence, Billing, OOP.

Introducción

Tecnologías Chapinas S.A. requiere automatizar la facturación de sus servicios de infraestructura en la nube, basados en configuraciones de recursos agrupadas por categorías. Este proyecto aborda dicha necesidad mediante el desarrollo de una solución integral compuesta por un API RESTful en Flask para el backend y una interfaz web simuladora en Django para el frontend. El sistema debe procesar archivos XML de configuración y consumo, persistir los datos en formato XML, utilizar POO para modelar el dominio y aplicar expresiones regulares para validaciones específicas. El propósito de este ensayo es documentar el diseño, implementación y funcionalidades del sistema desarrollado, demostrando el cumplimiento de los objetivos y requisitos establecidos.

Desarrollo del tema

El núcleo del proyecto radica en la creación de un sistema robusto y modular capaz de gestionar la información de la nube y calcular la facturación.

Arquitectura General

Se optó por una arquitectura desacoplada Frontend-Backend. El Backend, desarrollado con Flask, expone una API REST que centraliza toda la lógica de negocio y el acceso a los datos. El Frontend, construido con Django, actúa como un cliente de esta API, proporcionando una interfaz de usuario para interactuar con el sistema (carga de archivos, creación de datos, visualización, facturación). La comunicación entre ambos se realiza mediante peticiones HTTP, intercambiando datos principalmente en formato JSON para las operaciones interactivas, mientras que los archivos de carga masiva son XML.

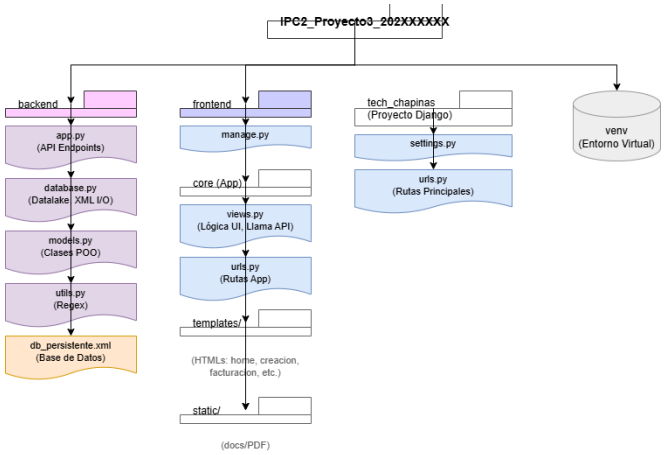
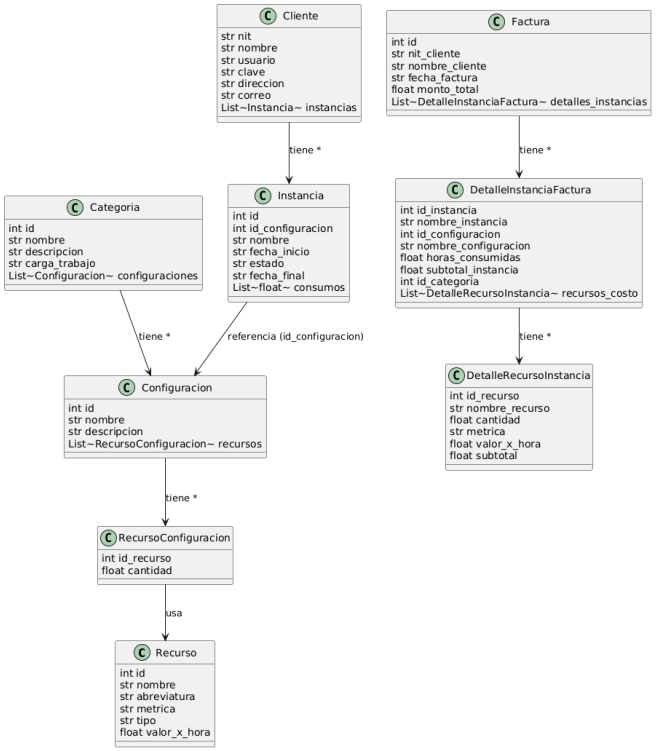


Figura 1. Arquitectura General del Sistema. Fuente: Elaboración propia.

Backend (API Flask)

El backend es el corazón del sistema y se organiza en los siguientes componentes principales:

- **Modelos POO (models.py):** Se definieron dataclasses para representar las entidades del dominio: Recurso, Categoria, Configuracion, Cliente, Instancia, y Factura (con sus detalles). Este enfoque POO facilita la manipulación estructurada de los datos y la encapsulación de la lógica.



FFigura 2. Diagrama de Clases simplificado del Backend. Fuente: Elaboración propia.

- **Gestor de Datos (database.py):** La clase Datalake actúa como una capa de abstracción sobre los datos. Es responsable de:
- Cargar datos desde los XML de configuración (cargar_desde_xml_string) y consumo (cargar_consumo_desde_xml_string).
- Mantener los datos en memoria (listas de objetos).
- Implementar la persistencia leyendo (cargar_desde_xml_persistente) y escribiendo (guardar_a_xml) el estado completo del sistema en el archivo db_persistente.xml, utilizando xml.etree.ElementTree para la construcción y xml.dom.minidom para el formateo legible ("pretty print").
- Proporcionar métodos de búsqueda (find_cliente, find_configuracion, etc.).
- **API Endpoints (app.py):** Se definieron rutas Flask para exponer las funcionalidades vía HTTP:
- /cargar-configuracion (POST, form-data): Recibe y procesa el XML de configuración.
- /cargar-consumo (POST, form-data): Recibe y procesa el XML de consumo.
- /crear-[entidad] (POST, JSON): Endpoints para crear clientes, recursos, categorías, configuraciones e instancias individualmente.
- /cancelar-instancia (POST, JSON): Modifica el estado de una instancia.
- /generar-factura (POST, JSON): Ejecuta el proceso de facturación para un cliente.
- /reporte/ventas-[tipo] (POST, JSON): Endpoints para obtener datos de reportes.
- /consultar-datos (GET): Devuelve el estado actual del sistema en JSON.
- /reset (POST): Limpia todos los datos.
- **Utilidades (utils.py):** Contiene funciones auxiliares, destacando:
- validar_nit(nit): Utiliza una expresión regular (r'(\d+-[\dkK])') para verificar el formato correcto del NIT.
- extraer_fecha(texto): Emplea una expresión regular (r'\b(0[1-9]|[12][0-9]|3[01])/(0[1-9]|1[0-2])/(\d{4})\b') para encontrar y extraer la primera fecha válida dd/mm/yyyy de una cadena de texto, descartando el resto.

Frontend (Interfaz Django)

La aplicación Django (core) sigue el patrón MVT (Modelo-Vista-Template) para ofrecer una interfaz web que consume la API del backend:

- **Vistas (views.py):** Contienen la lógica para interactuar con la API Flask usando la librería requests. Cada vista maneja las peticiones GET (mostrar formularios/datos) y POST (enviar datos/archivos al backend), procesa las respuestas JSON y pasa la información relevante a las plantillas. Se implementaron vistas para: home (carga de archivos y visualización de datos), creacion_datos_view (formularios para crear entidades), facturacion_view, reportes_view, reset_data, y ayuda_view. Se incluyó lógica para convertir fechas del formato YYYY-MM-DD (del input type="date") al formato dd/mm/yyyy esperado por el backend.
- **Plantillas (templates/core/*.html):** Usan el sistema de plantillas de Django con un base.html para la estructura común (menú lateral, estilos). Muestran los datos recuperados de la API (en tablas y formato JSON crudo en home.html), presentan formularios para la interacción del usuario (con validaciones básicas HTML5 y dropdowns dinámicos poblados con datos de la API), y muestran mensajes de éxito o error devueltos por el backend. Se utilizó JavaScript para implementar la funcionalidad de añadir recursos dinámicamente en el formulario de creación de configuraciones y para los dropdowns dependientes en el formulario de cancelación de instancia.

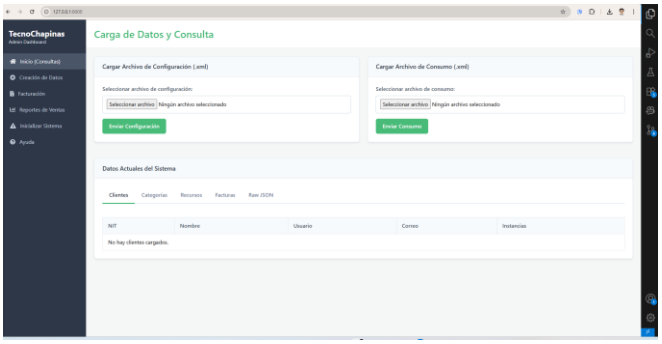


Figura 3. Ejemplo de la interfaz de usuario del Frontend. Fuente: Elaboración propia.

- **URLs (urls.py):** Mapean las URLs de la aplicación web a las vistas correspondientes.

Funcionalidades Clave Implementadas

Conclusiones

Se implementó un sistema de facturación para servicios en la nube con arquitectura desacoplada, utilizando Flask en el backend y Django en el frontend, aplicando POO, XML y expresiones regulares según los objetivos del proyecto.

La separación entre backend y frontend mediante una API REST permitió desarrollar y probar la lógica de negocio de forma independiente, destacando el uso de dataclasses para un modelado claro y mantenible.

La persistencia en XML cumplió con los requisitos, aunque implicó mayor complejidad frente a bases de datos tradicionales, mientras que la interfaz en Django ofreció funciones completas de carga, facturación y consulta con buena usabilidad.

Se propone como mejora implementar autenticación segura, manejo de errores más detallado, generación avanzada de reportes en PDF y posible migración a una base de datos relacional o NoSQL.

Referencias bibliográficas

Documentación Oficial de Flask. (s.f.). Recuperado de https://flask.palletsprojects.com/

Documentación Oficial de Django. (s.f.). Recuperado de https://docs.djangoproject.com/

Documentación Oficial de Python - xml.etree.ElementTree. (s.f.). Recuperado de https://docs.python.org/3/library/xml.etree.elementtree.html

W3Schools. (s.f.). XML Tutorial. Recuperado de https://www.w3schools.com/xml/

Real Python. (s.f.). Python Regular Expressions. Recuperado de https://realpython.com/regex-python/