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

DESARROLLO DE UN SISTEMA DE FACTURACIÓN PARA INFRAESTRUCTURA DE NUBE MEDIANTE API REST Y BASE DE DATOS XML

202300814 – Josue Geovany Yahir Perez Avendaño

Resumen

El presente ensayo describe el desarrollo de un sistema de facturación para servicios de infraestructura de nube implementado para la empresa Tecnologías Chapinas, S.A. El proyecto implementa una arquitectura cliente-servidor mediante API REST utilizando Flask como backend y Django para el frontend, almacenando la información en archivos XML. El sistema gestiona recursos de hardware y software, configuraciones de infraestructura organizadas por categorías, clientes e instancias aprovisionadas, así como el consumo de recursos y su correspondiente facturación detallada. La solución adopta el paradigma de programación orientada a objetos (POO) para modelar las entidades del sistema y utiliza expresiones regulares para la validación de datos. Los principales impactos técnicos incluyen la implementación de un sistema escalable que permite el seguimiento preciso de costos por recurso y la generación automática de facturas basadas en consumo real. Se concluye que la arquitectura propuesta permite una gestión eficiente y detallada de los servicios de nube, facilitando tanto la administración operativa como la transparencia en la facturación para los clientes.

El resumen y las palabras clave deberán ocupar únicamente esta columna.

Palabras clave
API REST, Infraestructura de nube, Facturación por consumo, Programación orientada a objetos, XML

Abstract

This essay describes the development of a billing system for cloud infrastructure services implemented for the company Tecnologías Chapinas, S.A. The project implements a client-server architecture through REST API using Flask as backend and Django for frontend, storing information in XML files. The system manages hardware and software resources, infrastructure configurations organized by categories, clients and provisioned instances, as well as resource consumption and its corresponding detailed billing. The solution adopts the object-oriented programming (OOP) paradigm to model system entities and uses regular expressions for data

validation. The main technical impacts include the implementation of a scalable system that allows precise cost tracking per resource and automatic invoice generation based on actual consumption. It is concluded that the proposed architecture enables efficient and detailed cloud service management, facilitating both operational administration and billing transparency for clients.

Keywords
REST API, Cloud infrastructure, Consumption-based billing, Object-oriented programming, XML.

Introducción

La computación en la nube ha revolucionado la forma en que las empresas aprovisionan y consumen recursos tecnológicos. En este contexto, surge la necesidad de desarrollar sistemas capaces de gestionar de manera precisa y transparente el uso de recursos computacionales y su correspondiente facturación. El presente ensayo expone el diseño e implementación de un sistema integral de facturación para servicios de infraestructura de nube, que permite a la empresa Tecnologías Chapinas, S.A. registrar, monitorear y facturar el consumo de recursos por parte de sus clientes.

El sistema aborda desafíos técnicos fundamentales como la gestión de múltiples tipos de recursos (núcleos de CPU, memoria RAM, almacenamiento, sistemas operativos, bases de datos), la organización de configuraciones de infraestructura según cargas de trabajo específicas, el registro detallado de consumos y la generación automática de facturas basadas en el uso real de recursos. La solución se construye sobre una arquitectura de servicios web RESTful que promueve la modularidad, escalabilidad y facilidad de integración con otros sistemas.

¿Cómo se puede implementar un sistema que facture detalladamente el consumo de recursos en infraestructura de nube? ¿Qué arquitectura de software es más apropiada para garantizar precisión, escalabilidad y mantenibilidad? Este ensayo responde estas interrogantes mediante el análisis de la arquitectura implementada, los modelos de datos utilizados y las decisiones de diseño adoptadas.

Desarrollo del tema

a. Arquitectura del Sistema

El sistema implementa una arquitectura cliente-servidor de tres capas que separa claramente las responsabilidades de presentación, lógica de negocio y persistencia de datos. La capa de presentación está desarrollada en Django siguiendo el patrón MVT (Modelo-Vista-Template), permitiendo una interfaz web interactiva para la gestión de recursos, clientes, instancias y procesos de facturación. La capa de lógica de negocio se implementa como una API RESTful utilizando Flask, exponiendo endpoints HTTP que procesan las operaciones del sistema. La capa de persistencia utiliza archivos XML como base de datos, proporcionando un formato estructurado y legible para almacenar la información.

Esta arquitectura desacoplada ofrece ventajas significativas en términos de mantenibilidad y escalabilidad. El frontend puede evolucionar independientemente del backend, e incluso podrían desarrollarse múltiples clientes (web, móvil, desktop)

que consuman la misma API. Durante el desarrollo, se utiliza versionamiento con Git y GitHub, permitiendo un control riguroso de cambios mediante la creación de cuatro releases progresivos que documentan la evolución del sistema.

b. Modelo de Dominio y Programación Orientada a Objetos

El sistema modela el dominio del problema mediante un conjunto de clases que representan las entidades fundamentales. La clase Recurso encapsula la información de un recurso de infraestructura (hardware o software), incluyendo atributos como identificador, nombre, abreviatura, métrica de medición, tipo y valor por hora. Cada recurso implementa un método calcular_costo() que determina el costo basándose en las horas de uso y la cantidad utilizada.

La clase Configuracion agrupa recursos específicos en cantidades determinadas, permitiendo crear paquetes de infraestructura preconfigurados. Cada configuración mantiene un diccionario de recursos donde la clave es el identificador del recurso y el valor es la cantidad incluida. El método calcular_costo_por_hora() itera sobre los recursos de la configuración para determinar el costo total por hora.

Las configuraciones se organizan en Categoria, que define el tipo de carga de trabajo para el cual están optimizadas (desarrollo, producción, económicas, alto rendimiento, etc.). Esta organización jerárquica facilita que los clientes seleccionen configuraciones apropiadas según sus necesidades específicas.

La clase Cliente almacena la información del cliente registrado, incluyendo NIT, nombre, credenciales de acceso, dirección y correo electrónico. Cada cliente puede tener múltiples instancias de configuraciones aprovisionadas, representadas por la clase Instancia. Una instancia vincula a un cliente con una configuración específica, manteniendo información sobre fechas de vigencia, estado (Vigente o Cancelada) y una lista de consumos registrados.

La clase Consumo registra cada evento de uso de una instancia, almacenando el tiempo consumido en horas y la fecha/hora del consumo. Cada consumo incluye un indicador booleano facturado que permite identificar qué consumos ya han sido incluidos en facturas generadas, evitando duplicación en la facturación.

Finalmente, la clase Factura representa el documento de cobro generado para un cliente en un período específico. Cada factura contiene múltiples objetos DetalleFactura, uno por cada instancia que tuvo consumos en el período facturado. El detalle desglosa el costo por recurso, proporcionando transparencia completa sobre los cargos aplicados.

Esta estructura orientada a objetos proporciona encapsulación de datos y comportamientos, facilitando la extensión futura del sistema (por ejemplo, agregando nuevos tipos de recursos o métodos de cálculo de costos) sin afectar el código existente.

c. Gestión de Datos con XML

El sistema utiliza XML como mecanismo de persistencia de datos, implementando un esquema estructurado que refleja las relaciones entre las entidades del dominio. La elección de XML proporciona legibilidad humana, facilidad de validación mediante esquemas XSD, y compatibilidad con múltiples plataformas y lenguajes de programación.

El XMLManager actúa como capa de abstracción entre el modelo de objetos y la persistencia XML, implementando operaciones CRUD (Create, Read, Update, Delete) para todas las entidades. Este componente utiliza la biblioteca xml.etree.ElementTree de Python para parsear, manipular y generar documentos XML. Cada clase del modelo implementa métodos to_xml_element() y from_xml_element() que permiten la serialización y deserialización bidireccional entre objetos Python y elementos XML.

El sistema mantiene múltiples archivos XML: uno para recursos, otro para categorías y configuraciones, uno para clientes e instancias, y uno para facturas generadas. Esta separación modular facilita el respaldo selectivo de datos y mejora el rendimiento en operaciones que solo requieren acceso a subconjuntos específicos de información.

d. Procesamiento de Mensajes de Entrada

El sistema recibe dos tipos de mensajes XML de entrada. El mensaje de configuración permite cargar masivamente recursos, categorías, configuraciones, clientes e instancias. El XMLConfigProcessor parsea este documento, valida la estructura y los datos, y crea los objetos correspondientes que son almacenados en la base de datos XML.

El mensaje de consumos notifica los tiempos de uso de instancias específicas. El XMLConsumoProcessor extrae esta información, valida que el cliente y la instancia existan, y registra el consumo asociado. Un aspecto crítico es que el sistema utiliza expresiones regulares para extraer fechas y horas del texto, permitiendo flexibilidad en el formato de entrada. El patrón \d{2}/\d{2}/\d{4} identifica fechas en formato dd/mm/yyyy, mientras que \d{2}/\d{2}/\d{4}\s+\d{2}:\d{2} captura fechas con hora.

Esta capacidad de procesamiento robusto de entrada es fundamental para la integración del sistema con otras plataformas que puedan generar datos en formatos diversos o con información adicional contextual.

e. Lógica de Facturación

El proceso de facturación constituye el núcleo funcional del sistema. El FacturacionService implementa el algoritmo que genera facturas para un rango de fechas especificado. El proceso itera sobre todos los clientes, y para cada cliente, analiza sus instancias buscando consumos no facturados dentro del período seleccionado.

Para cada instancia con consumos pendientes, el sistema calcula las horas totales consumidas sumando todos los registros de consumo en el rango de fechas. Posteriormente, obtiene la configuración asociada a la instancia y calcula el costo por cada recurso incluido en la configuración, multiplicando el costo por hora del recurso por la cantidad configurada y por las horas totales consumidas.

El sistema genera un DetalleFactura por instancia, que incluye el desglose de costos por recurso. Todos los detalles de un cliente se agrupan en una única Factura, que suma los costos totales y recibe un número único secuencial. Crucialmente, una vez generada la factura, todos los consumos procesados se marcan con facturado=True, garantizando que consumos no sean contabilizados en múltiples facturas.

Este diseño permite flexibilidad en la periodicidad de facturación (mensual, trimestral, bajo demanda) y

proporciona trazabilidad completa del origen de cada cargo en la factura.

f. Servicios RESTful

La API REST implementada con Flask expone endpoints organizados por recurso o funcionalidad. Los endpoints principales incluyen:

- **Recursos**: GET, POST, PUT, DELETE en /api/recursos para gestionar recursos individuales.
- **Categorías**: GET, POST, DELETE en /api/categorias para gestionar categorías y sus configuraciones.
- **Configuraciones**: GET, POST, PUT, DELETE en /api/categorias/{id}/configuraciones para gestionar configuraciones dentro de categorías.
- **Clientes**: GET, POST, DELETE en /api/clientes para gestionar clientes.
- **Instancias**: POST en /api/clientes/{nit}/instancias para aprovisionar instancias, PUT para cancelarlas.
- **Facturación**: POST en /api/facturacion/generar para generar facturas, GET para consultar facturas existentes.
- **Sistema**: POST en /api/sistema/inicializar para limpiar datos, POST en /api/sistema/cargar-configuracion y /api/sistema/cargar-consumos para procesar archivos XML.

Cada endpoint implementa validaciones apropiadas de datos de entrada, manejo de errores con códigos HTTP estándar (200 para éxito, 400 para errores de cliente, 404 para recursos no encontrados, 500 para errores del servidor), y respuestas JSON estructuradas con formato consistente que incluye indicadores de éxito, mensajes descriptivos y datos resultantes.

Esta API puede ser consumida no solo por el frontend Django desarrollado, sino también por otras herramientas como Postman, permitiendo pruebas automatizadas y potencial integración con sistemas externos.

g. Validaciones y Robustez

El sistema implementa múltiples capas de validación para garantizar la integridad de los datos. Las validaciones incluyen formatos de NIT mediante expresiones regulares que verifican la estructura dígitos-guion-dígito/K, validaciones de fechas y fechas con hora, verificación de tipos de recursos (Hardware o Software), validación de estados de instancias (Vigente o Cancelada), y verificación de valores numéricos positivos para costos y cantidades.

El módulo validators.py centraliza las funciones de validación, promoviendo reutilización de código y consistencia en las reglas de negocio aplicadas. Las validaciones se ejecutan tanto en el frontend (proporcionando retroalimentación inmediata al usuario) como en el backend (garantizando que datos inválidos no se persistan incluso si provienen de clientes alternativos).

El manejo de excepciones utiliza clases de excepción personalizadas como ValueError para errores de validación, permitiendo que las capas superiores capturen y respondan apropiadamente con mensajes de error informativos al usuario.

.

Conclusiones

El desarrollo del sistema de facturación para infraestructura de nube demuestra la efectividad de combinar arquitecturas de servicios RESTful con programación orientada a objetos y persistencia en XML para construir soluciones empresariales robustas y mantenibles. La separación clara entre frontend, backend y capa de datos facilita el desarrollo independiente de cada componente y permite escalabilidad futura.

La implementación de POO proporciona un modelo de dominio claro que encapsula tanto datos como comportamientos relacionados, facilitando la comprensión del sistema y su extensión. El uso de XML como formato de persistencia, aunque menos eficiente que bases de datos relacionales para grandes volúmenes, ofrece ventajas en portabilidad, legibilidad y facilidad de validación estructural.

El sistema logra el objetivo principal de proporcionar facturación detallada y precisa basada en el consumo real de recursos, ofreciendo transparencia completa a los clientes sobre los cargos aplicados. La arquitectura API permite que el sistema se integre fácilmente con otras plataformas y servicios, posicionándolo como componente central en un ecosistema más amplio de herramientas de gestión de nube.

Como reflexión final, este proyecto evidencia la importancia de aplicar principios sólidos de ingeniería de software (separación de responsabilidades, encapsulación, abstracción, modularidad) para construir sistemas que no solo funcionen correctamente, sino que sean mantenibles, extensibles y escalables a largo plazo. La experiencia adquirida en el diseño e implementación de APIs REST, modelado de dominios complejos y gestión de datos persistentes proporciona fundamentos valiosos aplicables a futuros proyectos de desarrollo de software empresarial.

Referencias bibliográficas

Fielding, R. T. (2000). Architectural Styles and the Design of Network-based Software Architectures. Doctoral dissertation, University of California, Irvine.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional.

Grinberg, M. (2018). Flask Web Development: Developing Web Applications with Python (2nd ed.). O'Reilly Media.

Lutz, M. (2013). Learning Python: Powerful Object-Oriented Programming (5th ed.). O'Reilly Media.

Percival, H., & Gregory, B. (2020). Architecture Patterns with Python. O'Reilly Media.