Escuela de Programación - Python C1

Trabajo Final: Sistema OdontoCare

Introducción

El objetivo principal de este proyecto es integrar los distintos contenidos del curso y aplicarlos al desarrollo de una solución backend completa y funcional. Para ello, el estudiante deberá implementar un sistema que combine los siguientes componentes fundamentales:

- **Framework Backend:** Desarrollo de una API REST utilizando **Flask**, organizada de forma profesional mediante **Blueprints** para asegurar modularidad y escalabilidad.
- **Persistencia de Datos:** Uso de una base de datos **SQLite**, gestionada a través de **SQLAlchemy** como ORM para modelar entidades, relaciones y operaciones CRUD.
- **Seguridad:** Implementación de un mecanismo de **autenticación basado en tokens**, garantizando el acceso seguro a los distintos recursos del sistema.
- **Cliente Externo:** Creación de un **script independiente en Python** que consuma los servicios de la API utilizando la biblioteca **requests**, demostrando la correcta interacción entre cliente y servidor.
- **Arquitectura Distribuida y Comunicación entre Servicios:** Crear imágenes en docker.

Este objetivo busca consolidar las competencias del nivel C1, permitiendo al estudiante demostrar su capacidad para diseñar, desarrollar e integrar un backend completo con enfoque profesional.

Escenario del Proyecto

Una red de clínicas dentales ha decidido modernizar sus operaciones creando una aplicación a la medida para gestionar las citas de los pacientes y la disponibilidad de los odontólogos. Actualmente, el sistema se maneja de forma manual, lo que provoca errores frecuentes, duplicidad de información y falta de trazabilidad en los procesos administrativos.

Como desarrollador backend asignado al proyecto, tu misión consiste en diseñar y construir una **solución integral, robusta y escalable**, que permita cubrir todas las necesidades del nuevo sistema de gestión **OdontoCare**. Para ello, se requiere el desarrollo de una **API RESTful profesional**, siguiendo buenas prácticas de arquitectura de software, seguridad y persistencia de datos.

El sistema debe permitir la administración eficiente de la información mediante los siguientes módulos esenciales:

- **Pacientes**
- **Doctores**
- **Centros médicos o clínicas**
- **Citas médicas**

Toda la información gestionada por la API debe persistir en una base de datos confiable. Además, el acceso a los recursos debe estar controlado mediante un mecanismo de **autenticación basado en tokens (JWT o similar)**, garantizando que solo los usuarios autorizados puedan interactuar con los datos.

El formato de comunicación de todos los servicios será exclusivamente **JSON**, por lo que cada endpoint debe responder consistentemente en este formato, tanto en operaciones exitosas como en manejo de errores.

El estudiante deberá definir y organizar adecuadamente la estructura del proyecto. Adicionalmente, deberá incluir un archivo **requirements.txt** para cada proyecto, en el cual se especifiquen todas las librerías y dependencias necesarias, incorporando aquellas que considere pertinentes para el correcto desarrollo de la actividad.

Objetivos principales del sistema

1. Diseñar una API RESTful organizada, modular y mantenible.
2. Implementar operaciones CRUD para pacientes, doctores, centros y citas.
3. Garantizar la persistencia de la información en una base de datos (SQL o NoSQL).
4. Incorporar un sistema de autenticación segura por tokens.
5. Asegurar que todas las respuestas se entreguen en formato JSON.
6. Aplicar buenas prácticas como validación de datos, manejo de excepciones, paginación y documentación básica del API.
7. Implementar una arquitectura distribuida basada en contenedores Docker.

Arquitectura de la Solución

Para garantizar un desarrollo ordenado, escalable y alineado con buenas prácticas de ingeniería de software, la API debe implementarse utilizando una **arquitectura modular basada en Blueprints de Flask**. Esto permitirá separar la lógica del sistema por dominios funcionales, facilitando su mantenimiento, comprensión y reutilización.

La solución **no debe concentrar todo el código en un solo archivo**. En su lugar, se exige una estructura organizada que distribuya la lógica en módulos claros y coherentes. La API deberá estructurarse, como mínimo, con los siguientes componentes:

auth_bp — Autenticación y Gestión de Usuarios

Encargado de todas las operaciones relacionadas con el acceso seguro al sistema. Debe incluir:

- Registro de usuarios autorizados.
- Inicio de sesión mediante validación de credenciales.
- Generación y validación de **tokens de autenticación (JWT)**.
- Manejo de errores de acceso.

Este módulo garantiza que todas las acciones dentro del sistema sean realizadas solo por usuarios autenticados.

admin_bp — Administración y Gestión de Centros, Pacientes y Doctores

Módulo orientado a tareas administrativas, encargado de configurar los elementos base del sistema. Debe incluir:

- Creación de entidades principales: centros médicos, pacientes y doctores.
- Carga de datos, tanto masiva como individual, utilizando archivos en formato JSON cuando sea requerido.
- Opciones de consulta para todos los tipos de registros, permitiendo:
    - Búsqueda individual por ID.
    - Visualización opcional de una lista completa de registros.

Este módulo está diseñado para usuarios con roles administrativos o de gestión.

citas_bp — Gestión Operativa de Citas

Responsable del núcleo funcional de OdontoCare: la planificación, administración y control de citas médicas. Debe incluir:

- Creación, actualización, consulta y eliminación de citas.
- Validación de disponibilidad de doctores y centros.
- Reglas operativas para evitar conflictos en la agenda.
- Respuestas en formato JSON con mensajes claros y estructurados.

Este módulo será el más utilizado durante la operación diaria del sistema.

Modelo de Datos (SQLAlchemy)

Debes definir al menos las siguientes tablas/modelos. Puede ser necesario la creación de estructuras adicionales al modelo esto queda a libertad del estudiante.

Usuario

- id_usuario (PK)
- username
- password
- rol (admin, medico, secretaria/o, paciente)

Paciente

Datos mínimos del paciente:

- id_paciente (PK)
- id_usuario (FK opcional)
- nombre
- teléfono
- estado(ACTIVO/INACTIVO)

Doctor

- id_doctor (PK)
- id_usuario (FK opcional)
- nombre
- especialidad

Centro Médico

- id_centro (PK)
- nombre
- direccion

Cita Médica

Relaciona paciente, doctor y centro:

- id_cita (PK)
- fecha
- motivo
- estado
- id_paciente (FK)
- id_doctor (FK)
- id_centro (FK)
- id_usuario_registra(FK)

Funcionalidades Detalladas

Autenticación y Seguridad

El sistema no puede confiar en un simple campo JSON.
Debe incluir:

- **POST /auth/login**
- Verificación de usuario y contraseña.
- Retorno de un token válido.
- Envío obligatorio del token en el header:
  Authorization: Bearer <token>

Todos los endpoints protegidos deben validar este token.

Carga Inicial de Datos — Múltiples Endpoints

El sistema debe permitir cargar datos desde un archivo local **CSV**, pero:

· El servidor NO recibe el archivo CSV.

· El cliente procesa el archivo y envía registro por registro.

Endpoints:

crear usuarios POST /admin/usuario

Rol requerido: Admin
Registra un usuario(y crea un usuario con rol "admin" o "secretaria").

crear doctores POST /admin/doctores

Rol requerido: Admin
Registra un doctor (y crea un usuario con rol "medico").

crear paciente POST /admin/pacientes

Rol requerido: Admin
Registra un paciente (y crea un usuario con rol "paciente").

crear centro POST /admin/centros

Rol requerido: Admin
Registra un centro médico.

Gestión de Citas (Core)

Las reglas de negocio deben consultar la base de datos. Puede ser necesario agregar otros métodos adicionales para cumplir la actividad, el estudiante deberá incluir aquellos endpoints adicionales.

Agendar Cita — POST /citas

- Roles: Cliente y Admin

· Validaciones obligatorias:

  - El **doctor** existe.
  - El **centro médico** existe.
  - El **paciente** existe y está activo.
  - **No se puede agendar una cita si el doctor ya tiene otra en la misma fecha y hora** (evitar doble reserva).

Listar Citas — GET /citas

- Doctor: solo ve sus propias citas.
- Secretaria: puede consultar citas filtrando por fecha.
- Admin: puede filtrar por doctor, centro, fecha, estado o paciente.
- Se usan query params para aplicar los filtros.

Cancelar Cita — PUT /citas/<id>

- Roles permitidos: Secretaria y Admin.

- Validaciones:

    - La cita existe.
    - No está cancelada.
    - Se cambia el estado a "Cancelada" y se devuelve un mensaje JSON confirmando la acción.

Archivo de Prueba (datos.csv)

El estudiante debe crear un archivo **datos.csv** con información ficticia que permita:

- Cargar doctores
- Cargar pacientes
- Cargar centros

Este archivo será leído por el cliente y enviado a la API.

Cliente Python

El estudiante debe entregar un script: **carga_inicial.py**, que:

1. Realice login con un usuario admin (del CSV).
2. Procese y envíe los registros del archivo **datos.csv**.
3. Cree una cita médica.
4. Imprima en consola el JSON con la cita creada.

Arquitectura Distribuida y Comunicación entre Servicios

Además de los módulos principales del sistema, la actividad requiere implementar una **arquitectura distribuida basada en contenedores Docker**, donde cada componente clave funcione como un **microservicio independiente**.
El estudiante deberá diseñar y desplegar el sistema OdontoCare como un conjunto de servicios autónomos, cada uno ejecutándose en su propio contenedor Docker y comunicándose **únicamente mediante servicios REST**, sin compartir bases de datos ni acceso directo entre ellos.

Los módulos se separan de la siguiente forma:

1. **Servicio de Gestión de Usuarios y Registro Administrativo**
2. **Servicio de Gestión de Citas**

El servicio de citas no debe acceder directamente a las bases de datos de los otros módulos. En su lugar, debe obtener toda la información necesaria únicamente mediante los **servicios REST** expuestos por los demás microservicios. Esto garantiza un intercambio de información adecuado, seguro y coherente con la arquitectura distribuida definida para la actividad.

El estudiante podrá realizar los ajustes necesarios en su diseño para cumplir con los requisitos de:

- Intercomunicación entre los diferentes servicios
- Independencia de bases de datos
- Aislamiento y responsabilidad única de cada módulo

Requisitos de Entrega y Demostración

La entrega final del proyecto no solo incluye el código, sino también la demostración práctica y la evidencia del funcionamiento del sistema de microservicios.

Código Funcional (fork Git)

El requisito fundamental es la entrega del código fuente completo y funcional.

- Plataforma: El código debe estar alojado en un repositorio Git.
- Contenido: Debe incluir todos los componentes del sistema OdontoCare, siguiendo la arquitectura distribuida definida (servicios de Usuarios y Citas).

El estudiante debe desarrollar y presentar un conjunto de *scripts* que demuestren de forma práctica el funcionamiento de los servicios.

Pruebas de Integración(Opcional)

Opcionalmente se pueden incluir o desarrollar pruebas de integración que validen la comunicación entre los servicios y el acceso externo a los *endpoints*.

Las pruebas de integración podrán incluir cualquiera de los siguientes métodos:

- *Scripts* que realicen llamadas directas a los *endpoints* del servicio (usando librerías HTTP o comandos como `curl`).
- Implementación de pruebas unitarias utilizando **unittest** o el módulo **flask.testing**.

Documentación de pruebas de Endpoints

Entrega de documentación o *scripts* con la siguiente información claramente indicada para cada prueba de *endpoint*:

- **El Endpoint Utilizado:** La ruta completa del servicio REST.
- **El Archivo de Entrada:** El cuerpo de la solicitud enviado, obligatoriamente en **formato JSON**.

Video Explicativo

Se requiere una demostración visual y concisa del aplicativo.

Requisito	Detalle

Duración Máxima	5 minutos
Contenido	Debe evidenciar claramente el funcionamiento completo del aplicativo, incluyendo la interacción entre los microservicios.
Funcionamiento	Mostrar el flujo de trabajo, desde la inicialización hasta la creación de una cita médica, destacando la comunicación RESTful.

Evaluación

La puntuación de cada pregunta es la siguiente:

- Pregunta 1: 100%

Se valorará la validez de la solución y la claridad de la argumentación.

El ejercicio tiene indicado en el enunciado su peso en la valoración final. Los criterios de evaluación para evaluar este ejercicio son los siguientes:

Pregunta	No logrado (C-)	Mínimamente logrado (C+)	Logrado (B)	Logrado de forma excelente (A)
Pregunta 1	La respuesta es incorrecta o no fue desarrollada	La respuesta es parcialmente correcta y está mínimamente justificada	La respuesta es correcta y describe ciertos pasos para llegar a la solución, y está justificada	La respuesta es correcta e indica todos los pasos para llegar a la solución. Se referencia correctamente el artículo y apuntes para justificar la respuesta

Formato

Se sugiere, con el objetivo de estandarizar el formato, la actividad siga las siguientes restricciones:

- El desarrollo del **video explicativo es obligatorio**; en caso contrario, la actividad será considerada **incompleta**. El video debe formar parte del proyecto alojado en Git.
- El ejercicio se desarrollará en la plataforma **Git**, en la cual se podrán realizar **múltiples entregas** durante el proceso de desarrollo.
- Una vez que el estudiante considere que el ejercicio está listo para su calificación, deberá **notificar al Docente Colaborador** mediante un mensaje que incluya la **ruta del repositorio** y el **nombre de usuario de Git**.
- Además, es obligatorio **incluir comentarios claros y adecuados en el código**, que faciliten su comprensión y mantenimiento.

Entrega

La entrega de la actividad deberá realizarse mediante un fork en GitHub.

> 💡 Únicamente se calificarán aquellos proyectos que hayan enviado el correo solicitando la revisión por parte del profesor colaborador de tu aula; en caso de no cumplir con este requisito y una vez finalizado el curso, **el estudiante suspenderá de manera automática el nivel**.