INFORME TÉCNICO

API REST DE AGENDA DE CONTACTOS

Desarrollada con FastAPI y SQLAlchemy

Proyecto:	Agenda FastAPI
Desarrollador:	Daniel
Fecha:	15 de October de 2025
Versión:	1.0.0
Tecnología Principal:	FastAPI + SQLAlchemy
Base de Datos:	SQLite (configurable para MySQL)

TABLA DE CONTENIDOS

1. RESUMEN EJECUTIVO

Este informe documenta el desarrollo de una API REST para gestión de agenda de contactos, implementada utilizando FastAPI como framework principal y SQLAlchemy como ORM para el manejo de base de datos. El proyecto implementa un sistema CRUD completo (Create, Read, Update, Delete) para la gestión de contactos personales, incluyendo validación de datos, manejo de errores y documentación automática de la API. **Características principales:**

• API REST totalmente funcional con endpoints CRUD
• Validación automática de datos con Pydantic
• Base de datos SQLite (configurable para MySQL/PostgreSQL)
• Documentación automática con Swagger UI y ReDoc
• Manejo robusto de errores y excepciones
• Arquitectura modular y escalable

Tecnologías utilizadas:

• FastAPI 0.104.1 - Framework web moderno para Python
• SQLAlchemy 2.0.23 - ORM para Python
• Pydantic 2.5.0 - Validación de datos
• Uvicorn 0.24.0 - Servidor ASGI
• SQLite - Base de datos embebida

2. DESCRIPCIÓN DEL PROYECTO

2.1 Objetivo del Proyecto
Desarrollar una API REST moderna y eficiente para la gestión de una agenda de contactos personales, permitiendo operaciones completas de creación, lectura, actualización y eliminación de registros de contactos. **2.2 Alcance Funcional**
El sistema permite:
• Registrar nuevos contactos con información completa
• Consultar contactos individuales o listas completas
• Actualizar información existente de contactos
• Eliminar contactos del sistema
• Validar unicidad de emails
• Manejar errores de manera consistente
2.3 Beneficios
• **Rapidez:** FastAPI es uno de los frameworks más rápidos disponibles
• **Documentación automática:** Swagger UI generado automáticamente
• **Validación robusta:** Pydantic garantiza integridad de datos
• **Escalabilidad:** Arquitectura modular fácil de extender
• **Estándares modernos:** Uso de type hints y async/await

3. ARQUITECTURA DEL SISTEMA

3.1 Patrón Arquitectónico

El proyecto sigue el patrón de arquitectura en capas con separación clara de responsabilidades:

Capa de Presentación (API Layer):

• main.py - Definición de endpoints y configuración de FastAPI
• Manejo de requests y responses HTTP
• Validación de entrada y serialización de salida

Capa de Lógica de Negocio (Business Layer):

• crud.py - Operaciones de negocio y reglas de validación
• Implementación de la lógica CRUD
• Manejo de excepciones de negocio

Capa de Acceso a Datos (Data Layer):

• models.py - Definición de modelos de base de datos
• database.py - Configuración de conexión a base de datos
• SQLAlchemy ORM para abstracción de base de datos

Capa de Schemas (Validation Layer):

• schemas.py - Modelos Pydantic para validación
• Serialización/deserialización de datos
• Documentación automática de API

3.2 Flujo de Datos

1. Request HTTP llega a FastAPI (main.py)
2. Validación automática con Pydantic schemas
3. Llamada a función CRUD correspondiente
4. Interacción con base de datos via SQLAlchemy
5. Respuesta serializada y enviada al cliente

4. ESTRUCTURA DEL CÓDIGO

4.1 Organización de Archivos

El proyecto está organizado de manera modular siguiendo mejores prácticas:

```
agenda_fastapi/ ■■■ app/ ■ ■■■ __init__.py # Inicialización del paquete ■ ■■■ main.py # Aplicación principal FastAPI ■ ■■■ database.py # Configuración de base de datos ■ ■■■ models.py # Modelos SQLAlchemy ■ ■■■ schemas.py # Esquemas Pydantic ■ ■■■ crud.py # Operaciones CRUD ■ ■■■ test_api.py # Script de pruebas ■■■ .env # Variables de entorno ■■■ requirements.txt # Dependencias ■■■ test_endpoints.sh # Script de pruebas bash ■■■ README.md # Documentación
```

4.2 Descripción de Componentes

main.py: Punto de entrada principal que define todos los endpoints de la API, configuración de FastAPI, manejo de excepciones y documentación automática. **models.py:** Define el modelo de datos 'Persona' usando SQLAlchemy con campos para id, nombre, apellido, email, teléfono, dirección y timestamps. **schemas.py:** Contiene los esquemas Pydantic para validación de entrada y serialización de salida: PersonaBase, PersonaCreate, PersonaUpdate y Persona. **crud.py:** Implementa todas las operaciones de base de datos incluyendo validación de reglas de negocio como unicidad de email. **database.py:** Configuración de conexión a base de datos, motor SQLAlchemy y gestión de sesiones.

5. ENDPOINTS DE LA API

Método	Endpoint	Descripción	Código Estado
GET	/	Mensaje de bienvenida	200
GET	/health	Health check	200
POST	/personas/	Crear nueva persona	201
GET	/personas/	Obtener todas las personas	200
GET	/personas/{id}	Obtener persona por ID	200
PUT	/personas/{id}	Actualizar persona	200
DELETE	/personas/{id}	Eliminar persona	204

5.1 Ejemplos de Uso

Crear una persona (POST /personas/):

```
{ "nombre": "Juan", "apellido": "Pérez", "email": "juan.perez@email.com",
"telefono": "+1234567890", "direccion": "Calle Principal 123" }
```

Respuesta exitosa (201 Created):

```
{ "id": 1, "nombre": "Juan", "apellido": "Pérez", "email":
"juan.perez@email.com", "telefono": "+1234567890", "direccion": "Calle
Principal 123", "created_at": "2024-01-01T10:00:00", "updated_at": null }
```

6. MODELOS DE DATOS

6.1 Modelo de Base de Datos (SQLAlchemy)

La tabla 'personas' contiene la siguiente estructura:

Campo	Tipo	Restricciones	Descripción
id	Integer	PRIMARY KEY, INDEX	Identificador único
nombre	String(100)	NOT NULL	Nombre de la persona
apellido	String(100)	NOT NULL	Apellido de la persona
email	String(100)	UNIQUE, INDEX, NOT NULL	Email único
telefono	String(20)	NULLABLE	Número de teléfono
direccion	Text	NULLABLE	Dirección completa
created_at	DateTime	SERVER DEFAULT	Fecha de creación
updated_at	DateTime	ON UPDATE	Fecha de actualización

6.2 Esquemas Pydantic

PersonaBase: Esquema base con campos comunes
PersonaCreate: Para creación (hereda de PersonaBase)
PersonaUpdate: Para actualizaciones (todos los campos opcionales)
Persona: Para respuestas (incluye id y timestamps)

6.3 Validaciones Implementadas

• Email único en el sistema
• Campos requeridos: nombre, apellido, email
• Validación automática de tipos de datos
• Longitud máxima de campos de texto

7. CONFIGURACIÓN Y DESPLIEGUE

7.1 Variables de Entorno

El archivo .env contiene la configuración del sistema:

```
DB_HOST=localhost DB_PORT=3306 DB_USER=agenda_user DB_PASSWORD=password123
DB_NAME=agenda_db DEBUG=True
```

7.2 Configuración Actual

• Base de datos: SQLite (desarrollo)
• Puerto: 8000
• Host: 0.0.0.0 (todas las interfaces)
• Reload: Activado (desarrollo)

7.3 Comando de Ejecución

Para ejecutar la aplicación desde el directorio agenda_fastapi:

```
PYTHONPATH=/home/daniel/FastApi/agenda_fastapi uvicorn app.main:app
--reload --host 0.0.0.0 --port 8000
```

7.4 URLs de Acceso

• API Base: http://localhost:8000
• Documentación Swagger: http://localhost:8000/docs
• Documentación ReDoc: http://localhost:8000/redoc
• OpenAPI Schema: http://localhost:8000/openapi.json

8. TESTING Y VALIDACIÓN

8.1 Scripts de Prueba Desarrollados
test_api.py: Script Python para pruebas automatizadas usando requests
• Prueba creación de personas
• Prueba obtención de listas
• Prueba obtención por ID
test_endpoints.sh: Script Bash completo para pruebas con curl
• Prueba todos los endpoints
• Verifica códigos de estado HTTP
• Formatea respuestas JSON
8.2 Casos de Prueba Implementados
1. Endpoint raíz (GET /) - Mensaje de bienvenida
2. Health check (GET /health) - Estado del sistema
3. Crear persona (POST /personas/) - Con datos válidos
4. Obtener personas (GET /personas/) - Lista completa
5. Obtener por ID (GET /personas/{id}) - Búsqueda específica
6. Actualizar persona (PUT /personas/{id}) - Modificación parcial
7. Eliminar persona (DELETE /personas/{id}) - Eliminación
8. Verificar eliminación - Confirmar borrado
8.3 Validaciones de Errores
• Email duplicado (400 Bad Request)
• Persona no encontrada (404 Not Found)
• Datos inválidos (422 Unprocessable Entity)
• Campos requeridos faltantes
8.4 Herramientas de Testing
• curl - Para pruebas manuales
• requests - Para pruebas automatizadas
• Swagger UI - Para pruebas interactivas
• Postman/Insomnia - Compatibles

9. ANÁLISIS DE CÓDIGO

9.1 Calidad del Código
Fortalezas Identificadas:
• Separación clara de responsabilidades
• Uso correcto de type hints de Python
• Manejo adecuado de excepciones
• Documentación automática con docstrings
• Validación robusta de datos
• Arquitectura modular y escalable
Patrones de Diseño Aplicados:
• Repository Pattern (crud.py)
• Dependency Injection (FastAPI)
• DTO Pattern (Pydantic schemas)
• Factory Pattern (database engine)
9.2 Métricas del Proyecto
• Total de archivos Python: 6
• Líneas de código aproximadas: 300+
• Endpoints implementados: 7
• Modelos de datos: 1 (Persona)
• Esquemas Pydantic: 4
• Operaciones CRUD: 5
9.3 Dependencias Utilizadas
• fastapi==0.104.1 - Framework principal
• uvicorn==0.24.0 - Servidor ASGI
• sqlalchemy==2.0.23 - ORM
• pydantic==2.5.0 - Validación
• python-dotenv==1.0.0 - Variables de entorno
• mysql-connector-python==8.2.0 - Conector MySQL
9.4 Compatibilidad
• Python 3.9+
• Bases de datos: SQLite, MySQL, PostgreSQL
• Sistemas operativos: Linux, Windows, macOS
• Contenedores: Docker compatible

10. CONCLUSIONES Y RECOMENDACIONES

10.1 Logros del Proyecto

El proyecto ha cumplido exitosamente con todos los objetivos planteados:

✓ **API Funcional:** Sistema CRUD completo y operativo
✓ **Documentación:** Swagger UI automático y documentación técnica
✓ **Validación:** Robusta validación de datos de entrada
✓ **Escalabilidad:** Arquitectura modular preparada para crecimiento
✓ **Testing:** Scripts de prueba automatizados
✓ **Configurabilidad:** Soporte para múltiples bases de datos

10.2 Beneficios Técnicos Obtenidos

• **Rendimiento:** FastAPI es uno de los frameworks más rápidos
• **Productividad:** Desarrollo ágil con documentación automática
• **Mantenibilidad:** Código limpio y bien estructurado
• **Robustez:** Manejo integral de errores y validaciones
• **Estándares:** Cumple con OpenAPI 3.0 y JSON Schema

10.3 Recomendaciones para Mejoras Futuras

Corto Plazo:

• Implementar autenticación y autorización (JWT)
• Añadir paginación avanzada con filtros
• Implementar logging estructurado
• Añadir validaciones adicionales (formato teléfono)
• Crear tests unitarios con pytest

Mediano Plazo:

• Migrar a PostgreSQL para producción
• Implementar cache con Redis
• Añadir rate limiting
• Implementar búsqueda avanzada
• Crear API versioning

Largo Plazo:

• Microservicios con múltiples APIs
• Implementar WebSockets para actualizaciones en tiempo real
• Añadir soporte para archivos adjuntos
• Dashboard de administración
• Integración con servicios externos

10.4 Conclusión Final

El proyecto FastAPI Agenda de Contactos representa una implementación sólida y profesional de una API REST moderna. Utiliza las mejores prácticas actuales de desarrollo en Python y está preparado para evolucionar hacia un sistema de mayor complejidad. La combinación de FastAPI, SQLAlchemy y Pydantic proporciona una base técnica robusta que garantiza escalabilidad, mantenibilidad y performance óptimo. El código desarrollado sirve como una excelente base para proyectos más complejos y demuestra competencias sólidas en desarrollo backend moderno con Python.