Manual Técnico Software: LAVAPP

Proyecto: SOFTWARE SPA MOTOS CARROS

Dirigido a desarrolladores, administradores de sistemas, y usuarios avanzados, el manual proporciona las instrucciones de ejecución, componentes y estructura del código del software LAVAPP. Su propósito es servir como referencia completa para la implementación, mantenimiento y uso eficiente del software LAVAPP.

Contenido

Introducción 3
Principales Componentes 3
1 Requisitos de sistema 3
1.1 Requisitos hardware: 3
1.2 Requisitos Software 4
1.3 Descarga del repositorio GitHub: 4
1.4 Paquetes Generales 4
1.5 Ejecución del programa 5
2 Estructura del Código 6
2.1 LavAppProject (Frontend en React) 6
2.2 LavAppProject (Backend python y FastAPI) 6
3 Funcionalidades Principales 7
3.1 Gestión de usuarios (Backend/Core/Models/UserModel.py) 7
3.2 Gestión Cliente (Backend/Core/Models/ClienteModel.py) 9
3.3 Gestión Vehículos (Backend/Core/Models/VehiculoModel.py) 10
3.4 Gestión Servicios Generales y Adicionales (Backend/Core/Models/ServicioModel.py) 11
3.5 Gestión Promociones (Backend/Core/Models/PromocionesModel.py) 13
3.6 Gestión Facturación (Backend/Core/Models/FacturaModel.py) 15
3.7 Gestión Configuración (backend/Core/Models/ConfigModel.py) 17
3.8 Gestión de Reportes (Backend/Core/Services/ReporteService.py) 18
3.9 Inicio de sesión (backend/Core/Services/AutenticacionService.py) 19
4 Manejo de Errores 20

Introducción

El desarrollo del software LAVAPP surge como una propuesta presentada durante el desarrollo de la asignatura Estructura de Datos del ITM, este desarrollo tiene como objetivo principal aplicar conceptos y técnicas de desarrollo aprendidos en la clase para diseñar y crear una solución a la temática de un software spa de carros y motos.

LAVAPP está desarrollado en lenguaje de programación Python como requisito para el Backend y React en la parte del Frontend, estos componentes se entienden como:

- Frontend: Una interfaz gráfica que permite a los usuarios visualizar de una manera amigable y fluida la funcionalidad del aplicativo
- Backend: Es el motor del sistema diseñado para procesar toda la lógica desarrollada

Esta documentación ofrece una descripción de cada componente, así como de los procedimientos de ejecución y componentes del código.

Principales Componentes

1. **Python**

El Software en el Backend está desarrollado en Python, permite ejecutar toda la lógica del programa, para procesar estructura de datos y operaciones como: crear, editar y eliminar usuarios y servicio almacenando esta información en base de datos.

2. **React**

La aplicación Frontend, desarrollada en React, proporciona a los usuarios una interfaz intuitiva para interactuar con LAVAPP.

3. **FastAPI**

FastAPI actúa como componente principal encargándose de enrutar todas las solicitudes a los servicios correspondiente garantizando la comunicaciones y flujo seguro entre el Frontend y el Backend.

1 Requisitos de sistema

Para implementar y ejecutar el software LAVAPP es necesario garantizar que el sistema tenga los requisitos adecuados en términos de software y hardware. A continuación, se detalla lo necesario:

1.1 Requisitos hardware:

- **CPU**: Procesador de al menos 4 núcleos. Se recomienda una CPU de 8 núcleos para manejar cargas elevadas y permitir el procesamiento en tiempo real.

- **RAM**: Mínimo de 8 GB de memoria RAM. Para un rendimiento óptimo en entornos de producción, se recomiendan 16 GB o más.
- **Almacenamiento**: Un mínimo de 50 GB de almacenamiento en disco. Se recomienda un SSD de 100 GB o superior para mejorar los tiempos de respuesta

1.2 Requisitos Software

- **Compatibilidad:** el programa solo es compatible con sistemas operativos Windows
- **Python:** Python mínimo 3.12.5
- **Node.js:** Mínimo versión 20

1.3 Descarga del repositorio GitHub:

Se debe contar con acceso a GitHub para acceder al repositorio del proyecto para clonar y descargar el archivo ZIP: https://github.com/Biltzaile/LavAppProject.git

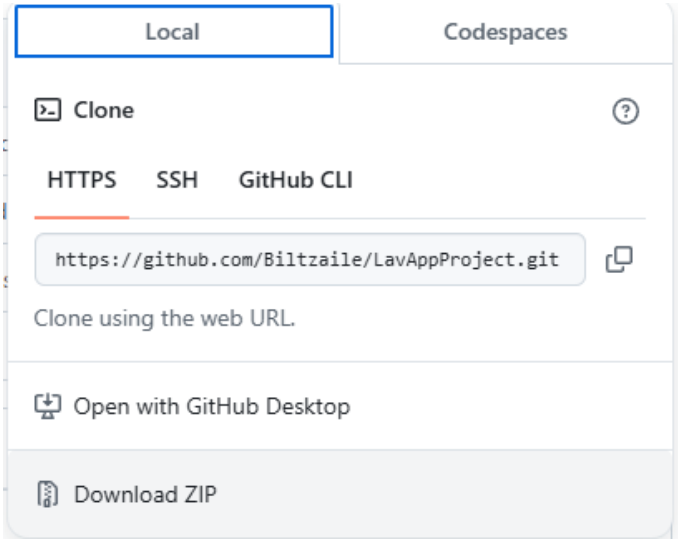


Imagen 1. Descarga del archivo zip del programa en GitHub

1.4 Paquetes Generales

Es necesario instalar varias herramientas y bibliotecas generales para facilitar la gestión del programa y el desarrollo de software.

Backend: Para instalar las librerías del archivo debe situarse dentro de la capeta Backend (Backend/requirements.txt) y ejecutar el siguiente comando en la terminal:

pip install -r requirements.txt

Frontend: Para instalar las librerías necesarias para el Front se ejecuta el siguiente comando, en la terminal de la IDE, debe situarse en la carpeta Frontend:

npm install

1.5 Ejecución del programa

Para la ejecución del Backend se debe abrir la terminar y ejecutar el siguiente comando:

uvicorn main:app --host 127.0.0.1 --port 8001 –reload ó uvicorn main:app

En el navegador se debe pegar la siguiente ruta: http://127.0.0.1:8001/docs

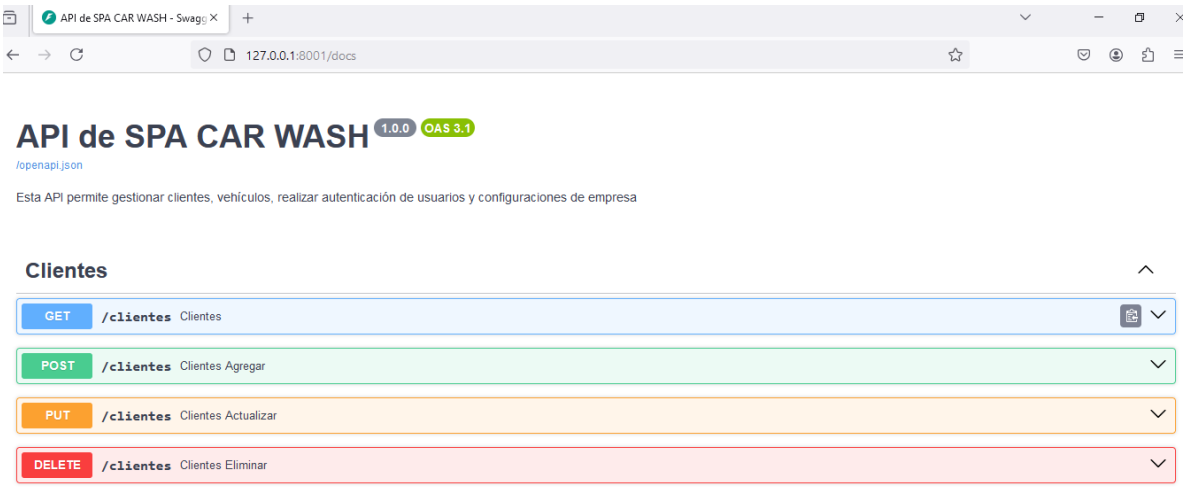


Imagen 2. Vista parcial de la API

- `GET /clientes/{cliente_id}`: Obtiene la información de un cliente por su ID.

- `POST /clientes/agregar/`: Agrega un nuevo cliente.

- `PUT /clientes/actualizar/{cliente_id}`: Actualiza la información de un cliente existente.

- `DELETE /clientes/eliminar/{cliente_id}`: Elimina un cliente por su ID.

Para la Ejecución del **Frontend** se debe abrir la terminar y ejecutar el siguiente comando:

npm run dev

2 Estructura del Código

La estructura del software LAVAPP es una estructura modular siguiendo la arquitectura hexagonal de puertos y adaptadores para un servicio de lavado de autos. Esta estructura permite comprender el programa y su gestión. A continuación, se describe la estructura de los principales módulos de LAVAPP y los componentes clave dentro de cada uno.

2.1 LavAppProject (Frontend en React)

Este módulo es la interfaz web que permite al usuario final gestionar la configuración de vehículos, usuarios, servicios y facturación dentro de la plataforma.

src/: Contiene el código fuente principal de la aplicación:

- **api**: contiene los servicios que hacen las llamadas a las APIs separadas por módulos para interactuar con el Backend
- **components**: contiene los componentes visuales
- **context**: contextos globales son utilizados para guardar datos de manera global en la APP
- **hooks**: funciones
- **lib y utils**: contiene funciones utilitarias que se utilizan en distintas partes del código
- **models**: interfaces y clases
- **pages**: contiene los archivos principales de cada módulo en cada uno se llaman los componentes que utilizan
- **routers**: allí están definidas las rutas de la app

package.json: contiene información, scripts de ejecución y lista de las dependencias o librerías que utiliza el proyecto.

2.2 LavAppProject (Backend python y FastAPI)

backend/spa_car_backend-main/: Carpeta principal del Código backend

- **main.py**: Archivo de arranque de la aplicación FastAPI.

Core:

- **Models**: contiene los modelos de datos de las estructuras de cada modulo
- **Services**: contiene los métodos de datos y validaciones de flujo de la estructura del modelo y conexiones con bases de datos.

DbContext: Entidades de base de datos que pertenecen a cada modulo

- clientes.csv
- config.json
- usuarios.csv
- vehículos.csv
- servicios_adicionales.csv
- servicios_generales.csv

Utilidades:

- config.py: el direccionamiento de ruta de conexión con base de datos
- responses.py: son respuestas y errores de las peticiones de la API en formato JSON

Web- API/ Controladores: Contiene los diferentes endpoint para las peticiones de los métodos del API

- Auth_controller.py
- Cliente_controller.py
- ConfigController.py
- Servicio_Controller.py
- Vehículo_Controller.py

requirements.txt: Lista las dependencias de Python necesarias para ejecutar el Backend.

3 Funcionalidades Principales

El software LAVAPP ofrece una variedad de funcionalidades. A continuación, se describen las principales funcionalidades con las partes del código y para que están implementadas:

3.1 Gestión de usuarios (Backend/Core/Models/UserModel.py)

El software LAVAPP permite a los usuarios registrar, crear, modificar y eliminar los usuarios según su propósito dentro de la aplicación, esta información se almacena en una base de datos.

UserModel.py: define dos modelos de datos utilizando la librería **Pydantic** en Python, que se utiliza para validar datos de manera eficiente y sencilla. Este modelo define la estructura y validaciones para crear un usuario. Asegura que los campos básicos como usuario, nombre, apellido, clave, y rol cumplan con los requisitos mínimos de longitud.

Funcionalidad de los Modelos

1. **Class UserModel**
   Este modelo define la estructura y validaciones para crear un usuario. Asegura que los campos básicos como usuario, nombre, apellido, clave, y rol cumplan con los requisitos mínimos de longitud.

- **Campos**:
  - usuario: Obligatorio, longitud entre 3 y 50 caracteres.
  - nombre: Obligatorio, longitud entre 3 y 50 caracteres.
  - apellido: Obligatorio, longitud entre 3 y 50 caracteres.
  - clave: Obligatoria, longitud mínima de 6 caracteres.
  - rol: Obligatorio, longitud entre 3 y 50 caracteres.

2. **class Usermodificar_facturaModel**
   Este modelo es una variación del anterior, diseñado para escenarios donde un usuario puede modificar sus datos. La principal diferencia es que el campo clave es opcional, permitiendo a los usuarios modificar información sin necesidad de proporcionar o cambiar la contraseña.
   - **Campos**:
     - usuario: Obligatorio, longitud entre 3 y 50 caracteres.
     - nombre: Obligatorio, longitud entre 3 y 50 caracteres.
     - apellido: Obligatorio, longitud entre 3 y 50 caracteres.
     - clave: Opcional, longitud mínima de 6 caracteres.
     - rol: Obligatorio, longitud entre 3 y 50 caracteres.

```python
from pydantic import BaseModel, Field

class UserModel(BaseModel):
    usuario: str = Field(..., min_length=3, max_length=50)
    nombre: str = Field(..., min_length=3, max_length=50)
    apellido: str = Field(..., min_length=3, max_length=50)
    clave: str = Field(..., min_length=6)
    rol: str = Field(..., min_length=3, max_length=50)

class UserUpdateModel(BaseModel):
    usuario: str = Field(..., min_length=3, max_length=50)
    nombre: str = Field(..., min_length=3, max_length=50)
    apellido: str = Field(..., min_length=3, max_length=50)
    clave: str | None = Field(default=None, min_length=6)
    rol: str = Field(..., min_length=3, max_length=50)
```

Imagen 3. Fragmento código modulo UserModel.py.

3.2 Gestión Cliente (Backend/Core/Models/ClienteModel.py)

El software LAVAPP permite a los usuarios registrar, crear, modificar y eliminar los clientes según su propósito dentro de la aplicación, esta información se almacena en una base de datos.

ClienteModel.py: el código tiene como uso principal validar, estructurar y transformar datos relacionados con clientes, asegurándose de que estos cumplan con ciertos requisitos antes de ser procesados o almacenados.

Descripción de los Campos y Validaciones

1. **Campos:**
    - tipo_doc: Tipo de documento del cliente. Debe ser uno de los valores permitidos (CC, NIT, CE, PP, TI o None).
    - documento: Número de documento. Obligatorio, con una longitud de entre 3 y 15 caracteres.
    - nombre: Nombre del cliente. Obligatorio, con una longitud de entre 1 y 50 caracteres.
    - apellido: Apellido del cliente. Obligatorio, con una longitud de entre 1 y 50 caracteres.
    - fec_nacimiento: Fecha de nacimiento. Debe ser una fecha válida y no posterior a la fecha actual.
    - telefono: Teléfono del cliente. Obligatorio, con una longitud de entre 7 y 15 caracteres.
    - email: Dirección de correo electrónico. Obligatoria, validada para tener un formato correcto y dominios específicos (.com, .co, .edu, .org).
2. **Validaciones Personalizadas:**
    - **Correo Electrónico (email_valido)**:
      Valida que el correo tenga un formato correcto y termine en uno de los dominios permitidos.
    - **Fecha de Nacimiento (parse_fecha_nacimiento y fecha_nacimiento_valida)**:
        - Convierte la fecha de nacimiento de texto al tipo date si es necesario.
        - Verifica que la fecha de nacimiento no sea posterior a la fecha actual.
3. **Método Adicional:**

- **to_dict**:
  Convierte una instancia del modelo en un diccionario, formateando la fecha de nacimiento como una cadena (YYYY-MM-DD).

```python
from pydantic import BaseModel, Field, field_validator,EmailStr
from datetime import date, datetime
from typing import Literal
import re

class ClienteModel(BaseModel):

    tipo_doc: Literal["CC","NIT","CE","PP","TI", None]
    documento: str = Field(..., min_length=3, max_length=15)
    nombre: str = Field(..., min_length=1, max_length=50)

    apellido: str = Field(..., min_length=1, max_length=50)
    fec_nacimiento: date
    telefono: str = Field(..., min_length=7, max_length=15)
    email: EmailStr


```

Imagen 4. Fragmento del código módulos ClienteModel.py

3.3 Gestión Vehículos (Backend/Core/Models/VehiculoModel.py)

El software LAVAPP permite a los usuarios registrar, crear, modificar y eliminar los vehículos según su propósito dentro de la aplicación, esta información se almacena en una base de datos.

VehiculoModel.py: El código tiene como propósito modelar y estructurar datos de vehículos, validando su formato y garantizando que cumplan con los requisitos establecidos, define restricciones para los campos.

Funcionalidad del Código

1. **Definición de Campos del Modelo:**
   - placa: La matrícula del vehículo. Debe tener una longitud entre 3 y 10 caracteres.
   - documento_cliente: Número de documento del cliente asociado al vehículo. Longitud entre 3 y 10 caracteres.
   - categoria: Categoría del vehículo, debe ser uno de los valores: "Moto", "Auto", "Cuatrimoto".

- segmento (opcional): Segmento del vehículo, como "Sedan", "SUV", etc.
- marca: Marca del vehículo, con una longitud mínima de 2 caracteres y máxima de 50.
- linea (opcional): Línea específica del modelo, como "Corolla" o "Civic".
- modelo: Año del modelo del vehículo. Debe estar entre 1900 y el año siguiente al actual.
- cilindrada: Tamaño del motor en centímetros cúbicos. Debe ser mayor a 0.
- grupo: Código o identificador del grupo del vehículo. Debe ser mayor a 0.

2. **Validaciones Incluidas:**
   - Los valores ingresados son validados automáticamente según las restricciones de cada campo (min_length, max_length, ge, le, etc.).
   - El año del modelo se válida para asegurarse de que sea válido dentro del rango permitido (1900 hasta el año siguiente al actual).

```python
from pydantic import BaseModel, Field
from typing import Literal, Optional

class VehiculoModel(BaseModel):
    placa: str = Field(..., min_length=3, max_length=10, alias="Placa")
    documento_cliente: str = Field(..., min_length=3, max_length=10,alias="Documento cliente")
    categoria: Optional[str] = Field (None ,alias="Categoria")
    segmento: Optional[str] = Field (None ,alias="Segmento")
    marca: str = Field(..., min_length=2, max_length=50,alias="Marca")
    linea: Optional[str] = Field (None ,alias="Linea")
    modelo: Optional[str] = Field (None ,alias="Modelo")
    cilindrada: int = Field(..., gt=0,alias="Cilindrada")
    grupo: Optional[str] = Field (None ,alias="Grupo")

    def to_dict(self): ...
            }

class Vehiculo: ...
        }
```

Imagen 5. Fragmento del código modulo VehiculoModel.py

3.4 Gestión Servicios Generales y Adicionales (Backend/Core/Models/ServicioModel.py)

El software LAVAPP permite a los usuarios registrar, crear, modificar y eliminar los servicios según su propósito dentro de la aplicación, esta información se almacena en una base de datos.

ServicioModel.py: El código modela y gestiona datos relacionados con servicios generales y servicios adicionales, para el manejo de categorías, precios y variables asociadas a servicios.

Descripción del Código

El código define dos modelos Pydantic para representar los datos y validaciones relacionadas con servicios en un sistema de gestión. Los modelos diferencian entre **servicios generales** y **servicios adicionales**, incorporando validaciones específicas y métodos para convertir los datos en un formato reutilizable.

Modelos y Funcionalidades

1. class GrupoValor

- Representa un grupo específico asociado a un valor de precio.
- Campos:
    - id: Identificador del grupo (entero).
    - precio: Precio asociado al grupo (flotante).

2. class CategoriaValor

- Representa una categoría de servicio que incluye múltiples grupos y sus valores.
- Campos:
    - categoria: Nombre de la categoría (cadena).
    - grupos: Lista de instancias del modelo GrupoValor.

3. class ServicioGeneralModel

- Modelo para los servicios generales.
- Campos:
    - id_servicio: Identificador único del servicio (opcional).
    - nombre: Nombre del servicio. Longitud mínima de 3 y máxima de 100 caracteres.
    - tipo_servicio: Debe ser "General" (valor literal).
    - valores: Lista de instancias de CategoriaValor, cada una representando categorías y sus grupos.
- Métodos:
    - **to_dict**: Convierte el modelo a un diccionario, transformando los precios en cadenas para facilitar el manejo.

4. class ServicioAdicionalModel

- Modelo para los servicios adicionales.
- Campos:
    - id_servicio: Identificador único del servicio (opcional).
    - nombre: Nombre del servicio. Longitud mínima de 3 y máxima de 100 caracteres.
    - tipo_servicio: Debe ser "Adicional" (valor literal).
    - categorias: Lista de categorías aplicables al servicio (lista de cadenas).
    - precio_variable: Indica si el precio depende de una variable (booleano).
    - variable: Unidad de medida si el precio es variable (ejemplo: "und", "m2").
    - precio_base: Precio base del servicio (flotante).
- Métodos:
    - **to_dict**: Convierte el modelo a un diccionario.

```python
from pydantic import BaseModel, Field
from typing import List, Optional, Literal

class GrupoValor(BaseModel):
    id: int
    precio: float

class CategoriaValor(BaseModel):
    categoria: str
    grupos: List[GrupoValor]

class ServicioGeneralModel(BaseModel):
    id_servicio: Optional[int] = 0
    nombre: str = Field(..., min_length=3, max_length=100)
    tipo_servicio: Literal["Adicional", "General"]
    valores: List[CategoriaValor]

    def to_dict(self): ...
        }

class ServicioAdicionalModel(BaseModel): ...
        }
```

Imagen 6. Fragmento del código ServicioModel.py

3.5 Gestión Promociones (Backend/Core/Models/PromocionesModel.py)

El software LAVAPP ofrece a los usuarios un servicio para gestionar promociones dentro del sistema de servicios y facturación.

PromocionesModel.py: El código tiene como función principal validar datos promocionales integrado al sistema de facturación de la aplicación, asegurando que los

datos puedan interactuar de manera fiable al momento de interactuar con los datos recibidos en el modelo.

class PromocionesModel:

1. **id_promocion (int):**
    - Identificador único de la promoción.
2. **descripcion (str):**
    - Breve descripción de la promoción.
    - Requiere entre 3 y 50 caracteres.
3. **fecha_inicio (date):**
    - Fecha en que la promoción comienza.
4. **fecha_fin (date):**
    - Fecha en que la promoción finaliza.
5. **porcentaje (float):**
    - Porcentaje de descuento asociado a la promoción.
6. **estado (bool):**
    - Indica si la promoción está activa o inactiva.

```python
from pydantic import BaseModel, Field, field_validator
from datetime import date, datetime

class PromocionesModel(BaseModel):
    id_promocion: int
    descripcion: str = Field(..., min_length=3, max_length=50)
    fecha_inicio: date
    fecha_fin: date
    porcentaje: float
    estado: bool

    @field_validator('estado')
    def validar_estado(cls, valor):
        if isinstance(valor, str):
            return valor.lower() == 'true'
        return bool(valor)

    @field_validator('fecha_inicio', 'fecha_fin')
    def parse_fecha(cls, valor): ...
        return valor
```

Imagen 7. Fragmento del código modulo PromocionesModel.py

3.6 Gestión Facturación (Backend/Core/Models/FacturaModel.py)

El software LAVAPP ofrece a los usuarios un servicio de facturación según su propósito dentro de la aplicación, tales como la validación y exportación de datos.

FacturaModel.py: El código tiene como función principal gestionar datos de una factura, como número, fecha, placa del vehículo, cliente, medio de pago, descuento y una lista de servicios.

Estructura de los Modelos

1. class ServicioFactura

Este modelo representa un servicio en una factura. Incluye:

- **id_servicio (int):**
  - Identificador del servicio.
  - Si no se proporciona, el valor predeterminado es 9999.
- **cantidad (int, opcional):**
  - Número de unidades del servicio (por defecto, 1).
- **descripcion (str):**
  - Detalle o nombre del servicio.
- **valor (float):**
  - Precio unitario del servicio.

Validador:

- validate_id_servicio: Si el campo id_servicio no está presente, se asigna 9999 como valor predeterminado.

2. class Factura

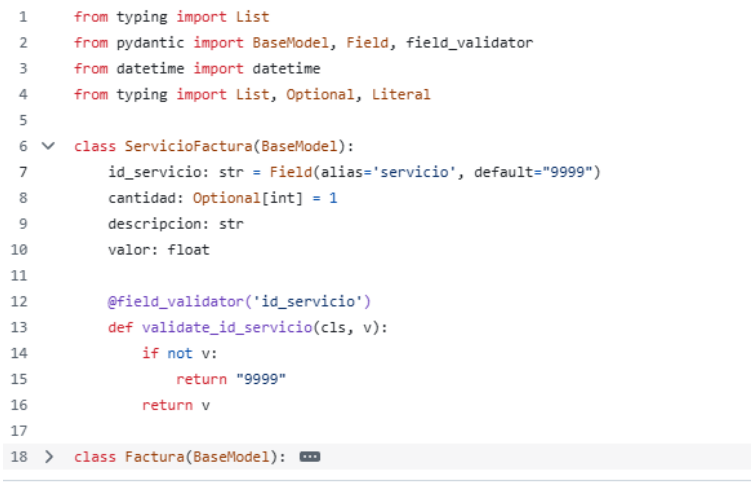
Modelo principal para representar una factura. Incluye:

- **numero_factura (str, opcional):**
  - Número de la factura. Si está ausente, se asigna 0 temporalmente.
- **fecha (datetime):**
  - Fecha de emisión de la factura.
- **placa (str):**
  - Placa del vehículo asociado.
- **categoria (str):**

    - Categoría del vehículo (e.g., "Auto", "Moto").
- **grupo (int):**
    - Grupo al que pertenece el vehículo para tarifas.
- **id_cliente (str):**
    - Identificador del cliente.
- **medio_pago (str):**
    - Método de pago utilizado (e.g., "Efectivo", "Tarjeta").
- **descuento (float):**
    - Porcentaje de descuento aplicado.
    - Validado para estar entre 0 y 100.
- **vlr_descuento (float):**
    - Valor monetario del descuento.
    - Validado para no ser negativo.
- **subtotal y total (float):**
    - Valores calculados antes y después del descuento, respectivamente.
- **servicios (List[ServicioFactura]):**
    - Lista de servicios asociados a la factura.

Validadores:

- validate_numero_factura: Si está ausente, asigna 0 temporalmente.
- validate_descuento: Verifica que el porcentaje esté entre 0 y 100.
- validate_vlr_descuento: Asegura que el valor no sea negativo. Métodos



```python
from typing import List
from pydantic import BaseModel, Field, field_validator
from datetime import datetime
from typing import List, Optional, Literal

class ServicioFactura(BaseModel):
    id_servicio: str = Field(alias='servicio', default="9999")
    cantidad: Optional[int] = 1
    descripcion: str
    valor: float

    @field_validator('id_servicio')
    def validate_id_servicio(cls, v):
        if not v:
            return "9999"
        return v

class Factura(BaseModel): ...
```

Imagen 7. Fragmento del código modulo FacturaModel.py

3.7 Gestión Configuración (Backend/Core/Models/ConfigModel.py)

El software LAVAPP permite a los usuarios configurar su entorno visual según su propósito dentro de la aplicación.

ConfigModel.py: El código tiene como función editar los datos del negocio y editar la apariencia de la interfaz.

1. class EmpresaModel

Modelo para representar los datos principales de la empresa.

- **nombre (str):** Nombre de la empresa.
- **nit (str):** Número de Identificación Tributaria (NIT) de la empresa.
- **telefono (str):** Número de contacto de la empresa.
- **direccion (str):** Dirección física de la empresa.
- **logo (str):** Imagen del logo, representada en formato Base64 o URL.

2. class TemaModel

Modelo que define el esquema de colores de la interfaz:

- **primario (str):** Color principal en formato HSL (Hue, Saturation, Lightness).
- **foregroundPrimario (str):** Color de primer plano asociado al color principal, también en formato HSL.

3. class ConfigModel

Modelo principal que combina las configuraciones de la empresa y el tema.

- **empresa (EmpresaModel):** Instancia del modelo de empresa.
- **tema (TemaModel):** Instancia del modelo de tema.

```python
# Core/Models/ConfigModel.py
from pydantic import BaseModel

class EmpresaModel(BaseModel):
    nombre: str
    nit: str
    telefono: str
    direccion: str
    logo: str

class TemaModel(BaseModel):
    primario: str
    secundario: str

class ConfigModel(BaseModel):
    empresa: EmpresaModel
    tema: TemaModel

    class Config:
        json_schema_extra = { ...
```

Imagen 9. Fragmento del código ConfigModel.py

3.8 Gestión de Reportes (Backend/Core/Services/ReporteService.py)

El software LAVAPP implementa un servicio de generación de los servicios realizados

ReporteService.py: Está diseñada para trabajar con un sistema de facturación y permite filtrar, buscar, y resumir datos de facturas mediante funciones específicas. Este módulo puede ser utilizado para generar reportes operativos, financieros y estadísticos en un sistema de gestión.

Funciones Principales

1. **get_all**
   Permite obtener todas las facturas registradas, con la opción de filtrar por un rango de fechas (fecha_inicio y fecha_fin). Las fechas deben estar en el formato YYYY-MM-DD.

2. **get_by_numero_factura**
   Busca y devuelve las facturas que coinciden con un número de factura específico.

3. **get_resumen**
   Genera un resumen de ventas que incluye:

   - Total, de ventas y número de facturas en el rango de fechas especificado.
   - Ventas agrupadas por medios de pago (TR, TD, TC, EF).
   - Resumen diario de ventas, incluyendo desgloses por categorías de vehículos (Moto, Auto, Cuatrimoto).

```python
from datetime import datetime
from typing import List, Dict
from Core.Models.FacturaModel import Factura
from Core.Services.FacturaServices import FacturaServices
from Core.Models.FacturaModel import Factura
import csv
import os

class ReporteServices:
    @classmethod
    def get_all(cls, fecha_inicio: str = None, fecha_fin: str = None):
        """
        Obtiene todas las facturas filtradas por rango de fecha.
        fecha_inicio y fecha_fin deben estar en formato 'YYYY-MM-DD'
        """
        try:
            # Obtener todas las facturas
            facturas = FacturaServices.get_all()

            if isinstance(facturas, str) and "Error" in facturas:
                return facturas

            # Si no hay fechas de filtro, retornar todas las facturas
            if not fecha_inicio or not fecha_fin:
                return facturas
```

Imagen 10. Fragmento del código modulo ReporteService.py

3.9 Inicio de sesión (backend/Core/Services/AutenticacionService.py)

El software LAVAPP implementa un servicio de autenticación y gestión de usuarios.

AutenticacionService.py: el código permite validar la credencial de acceso mediante un esquema de acceso hash de contraseña.

```python
from Core.Models.UserModel import UserModel
from utilidades import config
import csv
import hashlib
import os

class AuthService:
    users = []
    COLUMNAS_CSV = ['USUARIO', 'NOMBRE','APELLIDO','CLAVE', 'ROL']

    @classmethod
    def cargar_usuarios(cls): ...
            print(f"Error al leer el archivo: {e}")

    @classmethod
    def registrar_usuario(cls, user: UserModel): ...
            return f"Error al escribir en el archivo: {e}"

    @classmethod
    def verificar_credenciales(cls, usuario: str, clave: str): ...
        return None
```

Imagen 10. Fragmento del código AutenticacionService.py

4 Manejo de Errores

La API maneja diferente tipo de errores y devuelve respuestas adecuadas. Cada error se identifica y gestiona de manera adecuada, devolviendo respuestas estructuradas que permiten a los usuarios y desarrolladores entender y corregir problemas rápidamente.

- `404 Not Found`: Cuando no se encuentra la página solicitada.
- `400 Bad Request`: Cuando los datos proporcionados en la solicitud no son válidos.
- `500 Internal Server Error`: Cuando ocurre un error inesperado en el servidor.