

Documentación de Sistema web inteligente para la detección de enfermedades en la hoja de mango

1. Introducción

La agricultura moderna se enfrenta a desafíos significativos, incluyendo la gestión de enfermedades que afectan a los cultivos. El mango (*Mangifera indica*) es una fruta tropical de gran importancia económica a nivel mundial, pero su producción se ve constantemente amenazada por diversas enfermedades foliares. La detección temprana y precisa de estas enfermedades es crucial para implementar medidas de control efectivas y minimizar las pérdidas de rendimiento. Tradicionalmente, la identificación de enfermedades se ha basado en la inspección visual por parte de expertos, un proceso que puede ser lento, costoso y propenso a errores, especialmente en grandes extensiones de cultivo.

En respuesta a esta problemática, se ha desarrollado un Sistema Web Inteligente para la Detección de Enfermedades en la Hoja de Mango. Este sistema aprovecha los avances en la inteligencia artificial, específicamente el aprendizaje profundo y la visión por computadora, para ofrecer una solución automatizada y eficiente. Al integrar un backend robusto basado en Flask (Python), un frontend interactivo desarrollado con React (Vite JSX), una base de datos MySQL para la gestión de información y un modelo de TensorFlow pre-entrenado para la clasificación de imágenes, el sistema proporciona una herramienta integral para agricultores, agrónomos y operadores.

El objetivo principal de este documento es proporcionar una descripción técnica exhaustiva del sistema, abarcando su arquitectura, componentes clave, metodologías de implementación y el fundamento científico detrás de la detección de enfermedades. Se detallará cómo cada módulo contribuye al funcionamiento general del sistema, desde la carga de imágenes hasta la presentación de diagnósticos precisos y recomendaciones de tratamiento. Además, se incluirán fragmentos de código fuente relevantes para ilustrar la implementación de funcionalidades críticas, asegurando una comprensión profunda de la ingeniería y la ciencia que sustentan esta plataforma.

2. Resumen Ejecutivo

El Sistema Web Inteligente para la Detección de Enfermedades en la Hoja de Mango es una aplicación integral diseñada para diagnosticar automáticamente ocho tipos de enfermedades foliares del mango, además de identificar hojas saludables. La arquitectura del sistema se compone de tres capas principales: la capa de presentación (frontend), la capa de lógica de negocio y datos (backend) y la capa de inteligencia artificial.

El **Frontend**, construido con React y Vite JSX, ofrece una interfaz de usuario intuitiva y responsive que permite a los usuarios cargar imágenes de hojas de mango y recibir diagnósticos instantáneos. Incluye módulos para la gestión de usuarios (administradores, operadores, agricultores) y la visualización de resultados.

El **Backend**, desarrollado en Flask con Python, actúa como el cerebro del sistema. Gestiona las solicitudes del frontend, interactúa con la base de datos MySQL para almacenar información de usuarios y predicciones, y orquesta el proceso de análisis de imágenes a través del módulo de IA. Utiliza Flask-SQLAlchemy para la ORM y Flask-JWT-Extended para la autenticación y autorización de usuarios. La configuración del sistema es flexible y se maneja a través de variables de entorno.

La **Base de Datos MySQL** almacena datos cruciales como perfiles de usuario, historial de predicciones, registros del sistema y estadísticas. Su diseño relacional asegura la integridad y consistencia de los datos, facilitando la recuperación y el análisis de información.

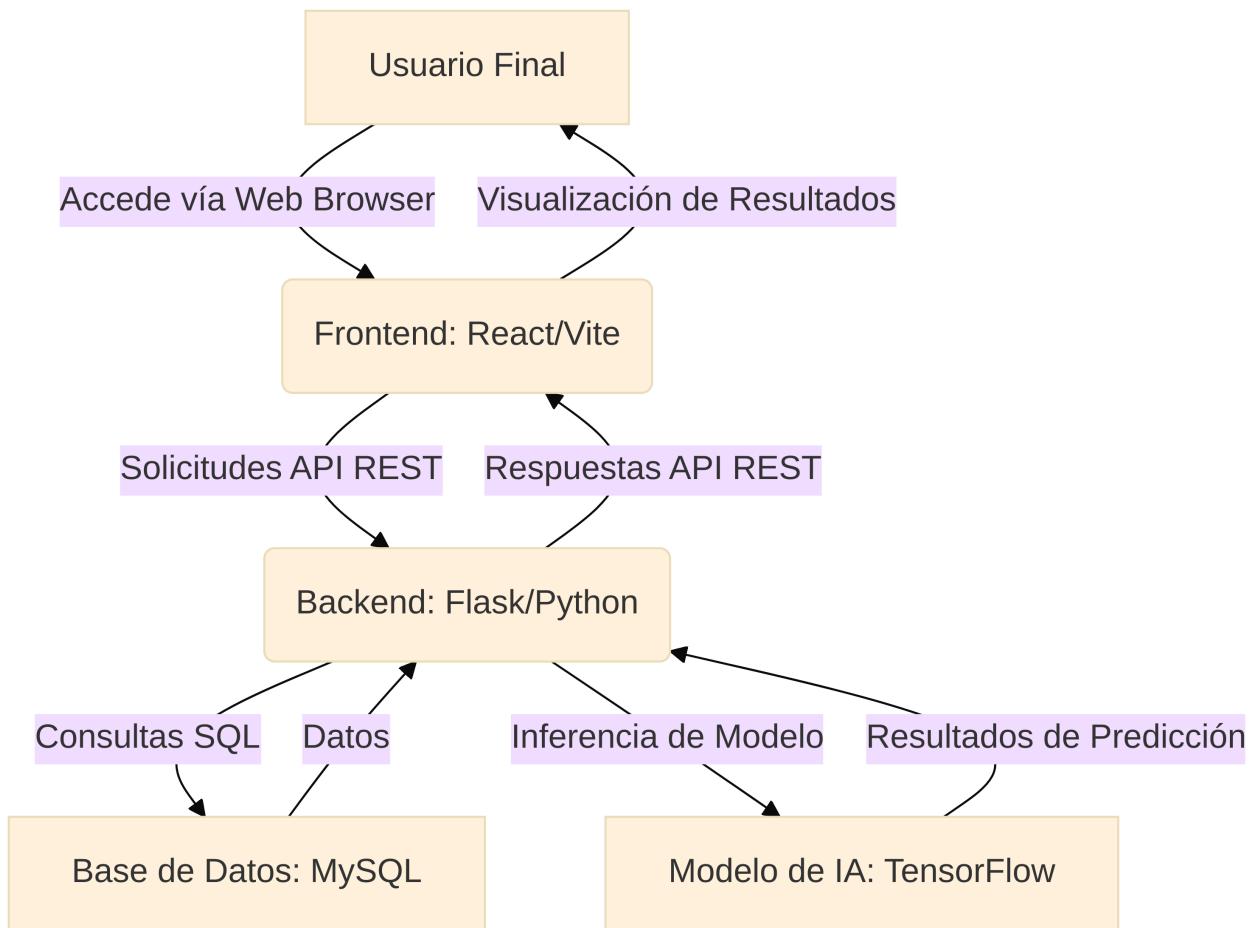
El componente central de **Inteligencia Artificial** es un modelo de aprendizaje profundo basado en TensorFlow (específicamente MobileNetV2), pre-entrenado para clasificar imágenes de hojas de mango en ocho categorías de enfermedades y una categoría saludable. Este modelo se integra como un servicio en el backend, permitiendo que las imágenes cargadas por los usuarios sean procesadas y analizadas para obtener un diagnóstico. El servicio de IA también proporciona información detallada sobre cada enfermedad, incluyendo síntomas, nombres científicos, tipos, tratamientos y prevenciones.

En conjunto, este sistema representa una solución tecnológica avanzada para la gestión fitosanitaria del cultivo de mango, promoviendo prácticas agrícolas más eficientes y sostenibles mediante la aplicación de la inteligencia artificial.

3. Arquitectura del Sistema

El Sistema Web Inteligente para la Detección de Enfermedades en la Hoja de Mango se adhiere a una arquitectura de tres capas, lo que permite una clara separación de responsabilidades, escalabilidad y mantenibilidad. Estas capas son: la capa de presentación (Frontend), la capa de lógica de negocio y datos (Backend), y la capa de inteligencia artificial, que se integra como un servicio dentro del backend.

3.1. Diagrama de Arquitectura General



Componentes Clave:

- **Usuario Final:** Interactúa con el sistema a través de un navegador web.
- **Frontend (React/Vite):** La interfaz de usuario que permite la interacción, carga de imágenes y visualización de resultados.
- **Backend (Flask/Python):** El servidor de aplicaciones que maneja la lógica de negocio, la autenticación, la gestión de datos y la integración con el modelo de IA.

- **Base de Datos (MySQL):** Almacena información de usuarios, historial de predicciones, logs del sistema y metadatos.
- **Modelo de IA (TensorFlow):** El componente central para la detección de enfermedades, responsable de clasificar las imágenes de hojas de mango.

3.2. Backend (Flask - Python)

El backend es el corazón del sistema, implementado con el microframework Flask en Python. Es responsable de la gestión de usuarios, la autenticación, la interacción con la base de datos y la orquestación del servicio de inteligencia artificial. La estructura de directorios del backend es la siguiente:

```
v7.0/backend/
├── app.py
├── app.log
├── config/
│   ├── __init__.py
│   └── settings.py
├── database/
│   ├── __init__.py
│   ├── connection.py
│   └── models.py
├── instance/
│   └── mango_disease.db
├── models/
│   └── mango_disease_model_mobilenetv2.h5
├── requirements.txt
├── routes/
│   ├── __init__.py
│   ├── admin_routes.py
│   ├── auth_routes.py
│   └── disease_routes.py
└── services/
    ├── __init__.py
    └── ai_service.py
└── setup_mysql.py
```

app.py : Este archivo es el punto de entrada principal de la aplicación Flask. Contiene la función `create_app()` que inicializa la aplicación, configura las extensiones (CORS, JWT, SQLAlchemy) y registra los blueprints de las rutas. También define rutas esenciales como `/health` para verificar el estado del sistema y `/api/info` para obtener información sobre la aplicación y el modelo de IA. Además, maneja los errores HTTP comunes.

```

# v7.0/backend/app.py

from flask import Flask, jsonify, request
from flask_cors import CORS
from flask_jwt_extended import JWTManager
from datetime import datetime
import logging
import os

from config import settings as config
from database.connection import init_mysql_database, mysql_manager
from services.ai_service import ai_service

# Configuración de logging
logging.basicConfig(level=logging.INFO, format=
    '%(asctime)s - %(name)s - %(levelname)s - %(message)s')
logger = logging.getLogger(__name__)

def create_app():
    app = Flask(__name__)

    # Cargar configuración desde settings.py
    app.config.from_object(config)

    # Configurar CORS
    CORS(app, resources={r"/api/*": {"origins": "*"}})

    # Inicializar JWT
    jwt = JWTManager(app)

    # Inicializar base de datos MySQL
    db = init_mysql_database(app)

    # Importar y registrar blueprints de rutas
    from routes.auth_routes import auth_bp
    from routes.disease_routes import disease_bp
    from routes.admin_routes import admin_bp

    app.register_blueprint(auth_bp, url_prefix="/api/auth")
    app.register_blueprint(disease_bp, url_prefix="/api/disease")
    app.register_blueprint(admin_bp, url_prefix="/api/admin")

    # Rutas de salud y información del sistema
    @app.route("/health", methods=["GET"])
    def health_check():
        """Verificar el estado de salud del sistema"""
        status = {
            "database": mysql_manager.test_connection(),
            "ai_model": ai_service.is_model_loaded()
        }
        overall_status = "OK" if all(status.values()) else "ERROR"

        try:
            return jsonify({
                "status": overall_status,
                "components": status,
                "timestamp": datetime.utcnow().isoformat()
            }), 200 if overall_status == "OK" else 503
        except Exception as e:
            logger.error(f"Error en health check: {e}")
            return jsonify({
                "status": "ERROR",

```

```

        "message": f"Error del sistema: {str(e)}",
        "timestamp": datetime.utcnow().isoformat()
    }), 500

@app.route("/api/info", methods=["GET"])
def get_system_info():
    """Obtener información del sistema"""
    try:
        model_info = ai_service.get_model_info()

        return jsonify({
            "app_name": config.APP_NAME,
            "version": config.APP_VERSION,
            "environment": config.ENVIRONMENT,
            "database": {
                "type": "MySQL",
                "host": config.DB_HOST,
                "port": config.DB_PORT,
                "database": config.DB_NAME
            },
            "ai_model": {
                "name": model_info["model_name"] if model_info else "No
disponible",
                "classes": model_info["classes"] if model_info else [],
                "total_classes": model_info["total_classes"] if model_info
else 0,
                "input_size": model_info["input_size"] if model_info else
"N/A"
            },
            "upload_limits": {
                "max_file_size_mb": config.MAX_FILE_SIZE_MB,
                "allowed_formats": config.ALLOWED_IMAGE_FORMATS
            },
            "server_info": {
                "host": config.SERVER_HOST,
                "port": config.SERVER_PORT,
                "debug": config.DEBUG
            }
        }), 200

    except Exception as e:
        logger.error(f"Error obteniendo info del sistema: {e}")
        return jsonify({"error": f"Error interno: {str(e)}"}), 500

# Manejo de errores
@app.errorhandler(404)
def not_found(error):
    return jsonify({
        "error": "Endpoint no encontrado",
        "message": "La ruta solicitada no existe",
        "available_endpoints": [
            "/api/auth/register",
            "/api/auth/login",
            "/api/disease/classes",
            "/api/disease/analyze-image",
            "/health",
            "/api/info"
        ]
    }), 404

@app.errorhandler(500)
def internal_error(error):
    logger.error(f"Error interno: {error}")
    return jsonify({

```

```

        "error": "Error interno del servidor",
        "message": "Ha ocurrido un error inesperado"
    }), 500

@app.errorhandler(413)
def too_large(error):
    return jsonify({
        "error": "Archivo demasiado grande",
        "message": f"El archivo excede el límite de
{config.MAX_FILE_SIZE_MB}MB"
    }), 413

@app.errorhandler(400)
def bad_request(error):
    return jsonify({
        "error": "Solicitud inválida",
        "message": "Los datos enviados no son válidos"
    }), 400

@app.errorhandler(401)
def unauthorized(error):
    return jsonify({
        "error": "No autorizado",
        "message": "Se requiere autenticación válida"
    }), 401

@app.errorhandler(403)
def forbidden(error):
    return jsonify({
        "error": "Acceso denegado",
        "message": "No tienes permisos para acceder a este recurso"
    }), 403

return app

def main():
    """Función principal"""
    print("🥭 SISTEMA DE DETECCIÓN DE ENFERMEDADES DE MANGO")
    print("=" * 70)
    print(f"🔧 Versión: {config.APP_VERSION}")
    print(f"🌐 Entorno: {config.ENVIRONMENT}")
    print(f"🗄️ Base de datos: MySQL ({config.DB_HOST}:
{config.DB_PORT}/{config.DB_NAME})")
    print("=" * 70)

    # Crear aplicación
    app = create_app()

    # Cargar modelo de IA
    try:
        ai_service.load_model()
        print("✅ Modelo de IA cargado exitosamente")
    except Exception as e:
        print(f"⚠️ Advertencia cargando modelo: {e}")
        print("💡 El sistema funcionará sin capacidades de IA")

        print(f"🌐 Servidor disponible en http://{config.SERVER_HOST}:
{config.SERVER_PORT}")
        print(f"📊 Panel de salud: http://{config.SERVER_HOST}:
{config.SERVER_PORT}/health")
        print(f"📋 Información del sistema: http://{config.SERVER_HOST}:
{config.SERVER_PORT}/api/info")
        print("=" * 70)
        print("🚀 Iniciando servidor...")

```

```
# Ejecutar servidor
try:
    app.run(
        host=config.SERVER_HOST,
        port=config.SERVER_PORT,
        debug=config.DEBUG,
        threaded=True
    )
except KeyboardInterrupt:
    print("\n⚠️ Servidor detenido por el usuario")
except Exception as e:
    print(f"\n❌ Error ejecutando servidor: {e}")

if __name__ == "__main__":
    main()
```

config/settings.py : Este archivo define todas las configuraciones importantes para la aplicación, como las credenciales de la base de datos, la clave secreta de JWT, los límites de tamaño de archivo para las cargas y los formatos de imagen permitidos. Centralizar la configuración en un solo lugar facilita la gestión y el despliegue en diferentes entornos.

```

# v7.0/backend/config/settings.py

import os

# Configuración de la aplicación
APP_NAME = "Mango Disease Detector"
APP_VERSION = "1.0.0"
ENVIRONMENT = os.getenv("FLASK_ENV", "development")
DEBUG = os.getenv("FLASK_DEBUG", "True").lower() == "true"

# Configuración del servidor
SERVER_HOST = os.getenv("SERVER_HOST", "0.0.0.0")
SERVER_PORT = int(os.getenv("SERVER_PORT", 5000))

# Configuración de la base de datos MySQL
DB_HOST = os.getenv("DB_HOST", "localhost")
DB_PORT = int(os.getenv("DB_PORT", 3306))
DB_USER = os.getenv("DB_USER", "root")
DB_PASSWORD = os.getenv("DB_PASSWORD", "password")
DB_NAME = os.getenv("DB_NAME", "mango_disease_db")

# Configuración de SQLAlchemy
SQLALCHEMY_DATABASE_URI = (
    f"mysql+pymysql://{DB_USER}:{DB_PASSWORD}@{DB_HOST}:{DB_PORT}/{DB_NAME}"
)
SQLALCHEMY_TRACK_MODIFICATIONS = False

# Configuración de JWT
JWT_SECRET_KEY = os.getenv("JWT_SECRET_KEY", "super-secret-jwt-key")
JWT_ACCESS_TOKEN_EXPIRES = int(os.getenv("JWT_ACCESS_TOKEN_EXPIRES", 3600)) # 1 hora

# Configuración de carga de imágenes
UPLOAD_FOLDER = os.getenv("UPLOAD_FOLDER", "uploads")
MAX_FILE_SIZE_MB = int(os.getenv("MAX_FILE_SIZE_MB", 5)) # 5 MB
MAX_CONTENT_LENGTH = MAX_FILE_SIZE_MB * 1024 * 1024 # Convertir MB a bytes
ALLOWED_IMAGE_FORMATS = ["jpeg", "jpg", "png"]

# Ruta del modelo de IA
AI_MODEL_PATH = os.getenv("AI_MODEL_PATH",
    "models/mango_disease_model_mobilenetv2.h5")

# Clases de enfermedades (orden importante para el modelo de IA)
DISEASE_CLASSES = [
    "antracnosis",
    "chancro bacteriano",
    "cortar gorgojo",
    "die back",
    "ball Midge",
    "saludable",
    "moho polvoriento",
    "moho hollín"
]
HEALTHY_CLASS_NAME = "saludable"

# Configuración de logging
LOG_FILE = os.getenv("LOG_FILE", "app.log")
LOG_LEVEL = os.getenv("LOG_LEVEL", "INFO")

# Roles de usuario
ROLES = {
    "administrador": 1,
    "operador": 2,
}

```

```
    "agricultor": 3
}

# Estadísticas iniciales del sistema
INITIAL_SYSTEM_STATS = {
    "total_users": 0,
    "total_predictions": 0,
    "last_backup": "N/A",
    "model_version": APP_VERSION
}
```

database/connection.py: Este módulo gestiona la conexión a la base de datos MySQL utilizando SQLAlchemy. Contiene la clase `MySQLManager` que encapsula la lógica para inicializar la base de datos, probar la conexión, obtener información de la base de datos, realizar copias de seguridad y registrar eventos del sistema. Es fundamental para la persistencia de datos y la monitorización.

```

# v7.0/backend/database/connection.py

from flask_sqlalchemy import SQLAlchemy
import mysql.connector
from mysql.connector import Error
import logging
from datetime import datetime
import os

from config import settings as config

logger = logging.getLogger(__name__)

db = SQLAlchemy()

class MySQLManager:
    def __init__(self):
        self.config = config

    def init_app(self, app):
        db.init_app(app)
        with app.app_context():
            db.create_all() # Crea las tablas si no existen
            self.initialize_system_stats() # Inicializa estadísticas del sistema

    def initialize_system_stats(self):
        from database.models import SystemStats # Importación local para evitar
        circularidad
        try:
            for stat_name, stat_value in self.config.INITIAL_SYSTEM_STATS.items():
                if not SystemStats.query.filter_by(stat_name=stat_name).first():
                    new_stat = SystemStats(stat_name=stat_name,
                                           stat_value=stat_value)
                    db.session.add(new_stat)

            db.session.commit()
            logger.info("Estadísticas del sistema inicializadas")
        except Exception as e:
            logger.error(f"Error inicializando estadísticas: {e}")
            db.session.rollback()

    def test_connection(self):
        """Probar conexión a MySQL"""
        try:
            connection = mysql.connector.connect(
                host=self.config.DB_HOST,
                port=self.config.DB_PORT,
                user=self.config.DB_USER,
                password=self.config.DB_PASSWORD,
                database=self.config.DB_NAME
            )

            if connection.is_connected():
                cursor = connection.cursor()
                cursor.execute("SELECT VERSION()")
                version = cursor.fetchone()
                logger.info(f"Conexión exitosa a MySQL {version[0]}")

                cursor.close()
                connection.close()
                return True

        except Error as e:

```

```

        logger.error(f"Error de conexión a MySQL: {e}")
        return False
    except Exception as e:
        logger.error(f"Error general de conexión: {e}")
        return False

    def get_database_info(self):
        """Obtener información de la base de datos"""
        try:
            info = {
                "type": "mysql",
                "host": self.config.DB_HOST,
                "port": self.config.DB_PORT,
                "database": self.config.DB_NAME,
                "tables": [],
                "users_count": 0,
                "predictions_count": 0,
                "logs_count": 0
            }

            # Obtener nombres de tablas
            inspector = db.inspect(db.engine)
            info["tables"] = inspector.get_table_names()

            # Contar registros
            from database.models import User, PredictionHistory, SystemLog # Importación local
            info["users_count"] = User.query.count()
            info["predictions_count"] = PredictionHistory.query.count()
            info["logs_count"] = SystemLog.query.count()

            return info
        except Exception as e:
            logger.error(f"Error obteniendo info de BD: {e}")
            return None

    def backup_database(self):
        """Crear backup de la base de datos MySQL"""
        try:
            backup_dir = "backups"
            os.makedirs(backup_dir, exist_ok=True)

            timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
            backup_file = os.path.join(backup_dir,
f"backup_mysql_{timestamp}.sql")

            # Comando mysqldump
            cmd = f"mysqldump -h {self.config.DB_HOST} -P {self.config.DB_PORT} -u {self.config.DB_USER} -p{self.config.DB_PASSWORD} {self.config.DB_NAME} > {backup_file}"

            result = os.system(cmd)
            if result == 0:
                logger.info(f"Backup MySQL creado: {backup_file}")

            # Actualizar estadística
            from database.models import SystemStats # Importación local
            backup_stat =
SystemStats.query.filter_by(stat_name="last_backup").first()
            if backup_stat:
                backup_stat.stat_value = datetime.utcnow().isoformat()
                db.session.commit()

            return backup_file

```

```

    else:
        logger.error("Error ejecutando mysqldump")
        return None

    except Exception as e:
        logger.error(f"Error en backup MySQL: {e}")
        return None

    def log_system_event(self, level, message, module=None, user_id=None,
ip_address=None, user_agent=None):
        """Registrar evento del sistema"""
        from database.models import SystemLog # Importación local
        try:
            log_entry = SystemLog(
                level=level,
                message=message,
                module=module,
                user_id=user_id,
                ip_address=ip_address,
                user_agent=user_agent
            )
            db.session.add(log_entry)
            db.session.commit()
        except Exception as e:
            logger.error(f"Error registrando log: {e}")
            db.session.rollback()

    def update_system_stat(self, stat_name, stat_value):
        """Actualizar estadística del sistema"""
        from database.models import SystemStats # Importación local
        try:
            stat = SystemStats.query.filter_by(stat_name=stat_name).first()
            if stat:
                stat.stat_value = str(stat_value)
            else:
                stat = SystemStats(stat_name=stat_name,
stat_value=str(stat_value))
                db.session.add(stat)

            db.session.commit()
        except Exception as e:
            logger.error(f"Error actualizando estadística: {e}")
            db.session.rollback()

mysql_manager = MySQLManager()

def init_mysql_database(app):
    """Función de conveniencia para inicializar MySQL"""
    mysql_manager.init_app(app)
    return mysql_manager

```

database/models.py: Este archivo define los modelos de la base de datos utilizando SQLAlchemy ORM. Incluye modelos para `User` (usuarios con diferentes roles), `PredictionHistory` (historial de predicciones de enfermedades), `SystemLog` (registros de eventos del sistema) y `SystemStats` (estadísticas generales del sistema). Estos modelos mapean las clases de Python a tablas en la base de datos MySQL.

```

# v7.0/backend/database/models.py

from datetime import datetime
from database.connection import db
from werkzeug.security import generate_password_hash, check_password_hash

class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(80), unique=True, nullable=False)
    email = db.Column(db.String(120), unique=True, nullable=False)
    password_hash = db.Column(db.String(128), nullable=False)
    role = db.Column(db.String(50), nullable=False, default='agricultor') #
    administrador, operador, agricultor
    created_at = db.Column(db.DateTime, default=datetime.utcnow)
    updated_at = db.Column(db.DateTime, default=datetime.utcnow,
    onupdate=datetime.utcnow)

    predictions = db.relationship('PredictionHistory', backref='user',
    lazy=True)
    logs = db.relationship('SystemLog', backref='user', lazy=True)

    def set_password(self, password):
        self.password_hash = generate_password_hash(password)

    def check_password(self, password):
        return check_password_hash(self.password_hash, password)

    def __repr__(self):
        return f'<User {self.username}>'

class PredictionHistory(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    user_id = db.Column(db.Integer, db.ForeignKey('user.id'), nullable=False)
    image_path = db.Column(db.String(255), nullable=False)
    predicted_class = db.Column(db.String(100), nullable=False)
    confidence = db.Column(db.Float, nullable=False)
    prediction_date = db.Column(db.DateTime, default=datetime.utcnow)
    # Opcional: almacenar las probabilidades de todas las clases
    all_class_confidences = db.Column(db.Text) # JSON string or similar

    def __repr__(self):
        return f'<Prediction {self.predicted_class} ({self.confidence:.2f})>'

class SystemLog(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    level = db.Column(db.String(20), nullable=False) # INFO, WARNING, ERROR,
    CRITICAL
    message = db.Column(db.Text, nullable=False)
    module = db.Column(db.String(100))
    timestamp = db.Column(db.DateTime, default=datetime.utcnow)
    user_id = db.Column(db.Integer, db.ForeignKey('user.id'), nullable=True)
    ip_address = db.Column(db.String(45)) # IPv4 o IPv6
    user_agent = db.Column(db.String(255))

    def __repr__(self):
        return f'<Log {self.level} - {self.message[:50]}>'

class SystemStats(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    stat_name = db.Column(db.String(100), unique=True, nullable=False)
    stat_value = db.Column(db.Text, nullable=False)
    last_updated = db.Column(db.DateTime, default=datetime.utcnow,
    onupdate=datetime.utcnow)

```

```
def __repr__(self):
    return f'<Stat {self.stat_name}: {self.stat_value}>'
```

routes/ : Este directorio contiene los blueprints de Flask que organizan las rutas de la API por funcionalidad. Esto mejora la modularidad y la organización del código.

- **auth_routes.py** : Maneja las rutas relacionadas con la autenticación de usuarios, incluyendo el registro (`/register`) y el inicio de sesión (`/login`). Utiliza JWT para generar tokens de acceso.

```
'''python
```

v7.0/backend/routes/auth_routes.py

```
from flask import Blueprint, request, jsonify from flask_jwt_extended import
create_access_token from database.models import User from database.connection
import db, mysql_manager from datetime import timedelta import logging

logger = logging.getLogger(name)

auth_bp = Blueprint('auth', name)

@auth_bp.route('/register', methods=['POST'])
def register():
    data = request.get_json()
    username = data.get('username')
    email = data.get('email')
    password = data.get('password')
    role = data.get('role', 'agricultor') # Default role
```

```

if not username or not email or not password:
    mysql_manager.log_system_event('WARNING', 'Intento de registro
fallido: datos incompletos', module='auth',
ip_address=request.remote_addr)
    return jsonify({'message': 'Faltan campos obligatorios'}), 400

if User.query.filter_by(username=username).first():
    mysql_manager.log_system_event('WARNING', f'Intento de registro
fallido: nombre de usuario {username} ya existe', module='auth',
ip_address=request.remote_addr)
    return jsonify({'message': 'El nombre de usuario ya existe'}), 409

if User.query.filter_by(email=email).first():
    mysql_manager.log_system_event('WARNING', f'Intento de registro
fallido: email {email} ya existe', module='auth',
ip_address=request.remote_addr)
    return jsonify({'message': 'El email ya está registrado'}), 409

new_user = User(username=username, email=email, role=role)
new_user.set_password(password)

try:
    db.session.add(new_user)
    db.session.commit()
    mysql_manager.log_system_event('INFO', f'Usuario {username} registrado
exitosamente con rol {role}', module='auth', user_id=new_user.id,
ip_address=request.remote_addr)
    mysql_manager.update_system_stat('total_users', User.query.count())
    return jsonify({'message': 'Usuario registrado exitosamente'}), 201
except Exception as e:
    db.session.rollback()
    logger.error(f'Error al registrar usuario: {e}', exc_info=True)
    mysql_manager.log_system_event('ERROR', f'Error al registrar usuario
{username}: {e}', module='auth', ip_address=request.remote_addr)
    return jsonify({'message': 'Error interno del servidor al registrar
usuario'}), 500

```

```

@auth_bp.route('/login', methods=['POST'])
def login():
    data = request.get_json()
    username = data.get('username')
    password = data.get('password')

```

```

user = User.query.filter_by(username=username).first()

if user and user.check_password(password):
    access_token = create_access_token(identity={'id': user.id,
                                              'username': user.username, 'role': user.role},
                                         expires_delta=timedelta(hours=1))
    mysql_manager.log_system_event('INFO', f'Usuario {username} inicio
sesión exitosamente', module='auth', user_id=user.id,
ip_address=request.remote_addr)
    return jsonify(access_token=access_token, user={'id': user.id,
                                              'username': user.username, 'role': user.role}), 200
else:
    mysql_manager.log_system_event('WARNING', f'Intento de inicio de
sesión fallido para usuario {username}', module='auth',
ip_address=request.remote_addr)
    return jsonify({'message': 'Credenciales inválidas'}), 401

```

```

- **disease\_routes.py**: Contiene las rutas para la detección de enfermedades, como la carga de imágenes para análisis (`/analyze-image`) y la obtención de la lista de clases de enfermedades (`/classes`).

```python

v7.0/backend/routes/disease_routes.py

```
from flask import Blueprint, request, jsonify from flask_jwt_extended import jwt_required, get_jwt_identity from services.ai_service import ai_service from database.connection import db, mysql_manager from database.models import PredictionHistory, User from werkzeug.utils import secure_filename import os import logging from config import settings as config

logger = logging.getLogger(name)

disease_bp = Blueprint('disease', name)

@disease_bp.route('/classes', methods=['GET']) @jwt_required() def get_disease_classes(): """Obtener la lista de clases de enfermedades que el modelo puede detectar."""
    try:
        model_info = ai_service.get_model_info()
        if model_info:
            return jsonify({'classes': model_info['classes']}), 200
        else:
            return jsonify({'message': 'Modelo de IA no cargado o información no disponible'}), 503
    except Exception as e:
        logger.error(f'Error al obtener clases de enfermedades: {e}')
        exc_info=True
    return jsonify({'message': 'Error interno del servidor'}), 500

@disease_bp.route('/analyze-image', methods=['POST']) @jwt_required() def analyze_image():
    """Analizar una imagen de hoja de mango para detectar enfermedades."""
    current_user = get_jwt_identity()
    user_id = current_user['id']
    username = current_user['username']
```

```

if \'file\' not in request.files:
    mysql_manager.log_system_event(\'WARNING\', \'Intento de análisis de
imagen sin archivo\', module=\'disease\', user_id=user_id,
ip_address=request.remote_addr)
    return jsonify({\'message\': \'No se encontró el archivo en la
solicitud\'}), 400

file = request.files[\'file\']

if file.filename == \'\':
    mysql_manager.log_system_event(\'WARNING\', \'Intento de análisis de
imagen con nombre de archivo vacío\', module=\'disease\', user_id=user_id,
ip_address=request.remote_addr)
    return jsonify({\'message\': \'Nombre de archivo no válido\'}), 400

if file and ai_service.allowed_file(file.filename):
    try:
        # Validar la imagen antes de guardarla
        ai_service.validate_image_content(file)
        file.seek(0) # Resetear el puntero del archivo después de la
validación

        filename = secure_filename(file.filename)
        # Crear directorio de subidas si no existe
        upload_dir = os.path.join(os.getcwd(), config.UPLOAD_FOLDER)
        os.makedirs(upload_dir, exist_ok=True)

        filepath = os.path.join(upload_dir, filename)
        file.save(filepath)
        logger.info(f\'Archivo guardado temporalmente en {filepath}\')

        prediction_result = ai_service.predict_disease(filepath)
        os.remove(filepath) # Eliminar archivo temporal
        logger.info(f\'Archivo temporal {filepath} eliminado\')

        if prediction_result:
            predicted_class = prediction_result[\'predicted_class\']
            confidence = prediction_result[\'confidence\']
            all_confidences = prediction_result[\'all_class_confidences\']
            disease_info = ai_service.get_disease_info(predicted_class)

            # Guardar historial de predicción
            new_prediction = PredictionHistory(
                user_id=user_id,
                image_path=filename, # Guardamos solo el nombre, no la ruta
completa
                predicted_class=predicted_class,
                confidence=confidence,
                all_class_confidences=str(all_confidences) # Convertir a
string para guardar en Text
            )
            db.session.add(new_prediction)
            db.session.commit()
            mysql_manager.log_system_event(\'INFO\', f\'Predicción realizada:
{predicted_class} con {confidence:.2f} de confianza\', module=\'disease\'',
user_id=user_id, ip_address=request.remote_addr)
            mysql_manager.update_system_stat(\'total_predictions\'',
PredictionHistory.query.count())

            return jsonify({
                \'predicted_class\': predicted_class,
                \'confidence\': confidence,
                \'disease_info\': disease_info,

```

```

        \\'all_class_confidences\': all_confidences
    }), 200
else:
    mysql_manager.log_system_event(\\'ERROR\', \\'Error en la
predicción del modelo de IA\', module=\\'disease\', user_id=user_id,
ip_address=request.remote_addr)
    return jsonify({\'message\': \'Error al procesar la imagen con el
modelo de IA\'}), 500
except ValueError as ve:
    mysql_manager.log_system_event(\\'WARNING\', f\'Error de validación de
imagen: {ve}\', module=\\'disease\', user_id=user_id,
ip_address=request.remote_addr)
    return jsonify({\'message\': str(ve)}), 400
except Exception as e:
    logger.error(f\'Error al analizar imagen: {e}\', exc_info=True)
    mysql_manager.log_system_event(\\'ERROR\', f\'Error interno al
analizar imagen: {e}\', module=\\'disease\', user_id=user_id,
ip_address=request.remote_addr)
    return jsonify({\'message\': \'Error interno del servidor al analizar
imagen\'}), 500
else:
    mysql_manager.log_system_event(\\'WARNING\', \\'Intento de análisis de
imagen con formato no permitido\', module=\\'disease\', user_id=user_id,
ip_address=request.remote_addr)
    return jsonify({\'message\': \'Formato de archivo no permitido\'}), 400

```

```

- **admin\_routes.py** : Proporciona rutas para funcionalidades administrativas, como la gestión de usuarios, la obtención de estadísticas del sistema y la realización de copias de seguridad de la base de datos. Estas rutas suelen requerir permisos de administrador.

```python

v7.0/backend/routes/admin_routes.py

```

from flask import Blueprint, jsonify, request from flask_jwt_extended import
jwt_required, get_jwt_identity from database.models import User,
PredictionHistory, SystemLog, SystemStats from database.connection import db,
mysql_manager from config.settings import ROLES import logging

logger = logging.getLogger(name)
admin_bp = Blueprint(\\'admin\', name)
def admin_required(fn): """Decorador para proteger rutas que solo pueden ser
accedidas por administradores.""" @jwt_required() def wrapper(args, kwargs):

```

```
current_user = get_jwt_identity() if current_user['role'] != 'administrador':  
    mysql_manager.log_system_event('WARNING', f'Acceso denegado a ruta de  
administrador para usuario {current_user["username"]} con rol  
{current_user["role"]}', module='admin', user_id=current_user['id'],  
ip_address=request.remote_addr) return jsonify({'message': 'Acceso denegado:  
Se requiere rol de administrador'}), 403  
return fn(args, **kwargs)  
return wrapper
```

```
@admin_bp.route('/users', methods=['GET']) @admin_required def get_users():  
    """Obtener todos los usuarios registrados.""" try: users = User.query.all() users_data  
= [{ 'id': user.id, 'username': user.username, 'email': user.email, 'role':  
user.role, 'created_at': user.created_at.isoformat(), 'updated_at':  
user.updated_at.isoformat() } for user in users]  
    mysql_manager.log_system_event('INFO', 'Acceso a lista de usuarios por  
administrador', module='admin', user_id=get_jwt_identity()['id'],  
ip_address=request.remote_addr) return jsonify(users_data), 200 except Exception  
as e: logger.error(f'Error al obtener usuarios: {e}', exc_info=True)  
    mysql_manager.log_system_event('ERROR', f'Error al obtener usuarios: {e}',  
module='admin', user_id=get_jwt_identity()['id'],  
ip_address=request.remote_addr) return jsonify({'message': 'Error interno del  
servidor'}), 500
```

```
@admin_bp.route('/users', methods=['PUT']) @admin_required def  
update_user(user_id): """Actualizar información de un usuario específico.""" user =  
User.query.get(user_id) if not user: mysql_manager.log_system_event('WARNING',  
f'Intento de actualizar usuario no existente: {user_id}', module='admin',  
user_id=get_jwt_identity()['id'], ip_address=request.remote_addr) return  
jsonify({'message': 'Usuario no encontrado'}), 404
```

```

data = request.get_json()
user.username = data.get('\username\', user.username)
user.email = data.get('\email\', user.email)
new_password = data.get('\password\'')
if new_password:
    user.set_password(new_password)
user.role = data.get('\role\', user.role)

# Validar que el rol sea uno de los permitidos
if user.role not in ROLES:
    mysql_manager.log_system_event('\WARNING\', f\Intento de asignar rol
inválido {user.role} a usuario {user_id}\', module=\admin\', user_id=get_jwt_identity()[\id\'], ip_address=request.remote_addr)
    return jsonify({\message\': \'Rol de usuario no válido\'}), 400

try:
    db.session.commit()
    mysql_manager.log_system_event('\INFO\', f\Usuario {user.username}
({user_id}) actualizado por administrador\', module=\admin\', user_id=get_jwt_identity()[\id\'], ip_address=request.remote_addr)
    return jsonify({\message\': \'Usuario actualizado exitosamente\'}), 200
except Exception as e:
    db.session.rollback()
    logger.error(f\Error al actualizar usuario {user_id}: {e}\', exc_info=True)
    mysql_manager.log_system_event('\ERROR\', f\Error al actualizar usuario
{user_id}: {e}\', module=\admin\', user_id=get_jwt_identity()[\id\'], ip_address=request.remote_addr)
    return jsonify({\message\': \'Error interno del servidor\'}), 500

```

```

@admin_bp.route('/users/\', methods=['DELETE\']) @admin_required def
delete_user(user_id): """Eliminar un usuario específico.""" user =
User.query.get(user_id) if not user: mysql_manager.log_system_event('\WARNING\' f\Intento de eliminar usuario no existente: {user_id}\', module=\admin\', user_id=get_jwt_identity()[\id\'], ip_address=request.remote_addr) return
jsonify({\message\': \'Usuario no encontrado\'}), 404

```

```

try:
    db.session.delete(user)
    db.session.commit()
    mysql_manager.log_system_event('\INFO\', f\Usuario {user.username}
({user_id}) eliminado por administrador\', module=\admin\', user_id=get_jwt_identity()[\id\'], ip_address=request.remote_addr)
    mysql_manager.update_system_stat('total_users', User.query.count())
    return jsonify({\message\': \'Usuario eliminado exitosamente\'}), 200
except Exception as e:
    db.session.rollback()
    logger.error(f\Error al eliminar usuario {user_id}: {e}\', exc_info=True)
    mysql_manager.log_system_event('\ERROR\', f\Error al eliminar usuario
{user_id}: {e}\', module=\admin\', user_id=get_jwt_identity()[\id\'], ip_address=request.remote_addr)
    return jsonify({\message\': \'Error interno del servidor\'}), 500

```

```
@admin_bp.route('/predictions', methods=['GET']) @admin_required def get_all_predictions(): """Obtener todas las predicciones realizadas en el sistema."""
try: predictions = PredictionHistory.query.all() predictions_data = [{\id\': p.id, \user_id\': p.user_id, \username\': p.user.username if p.user else \'N/A\', \image_path\': p.image_path, \predicted_class\': p.predicted_class, \confidence\': p.confidence, \prediction_date\': p.prediction_date.isoformat()} for p in predictions] mysql_manager.log_system_event('INFO', \'Acceso a historial de predicciones por administrador\', module='admin', user_id=get_jwt_identity()[\id\'], ip_address=request.remote_addr) return jsonify(predictions_data), 200 except Exception as e: logger.error(f\'Error al obtener predicciones: {e}\', exc_info=True) mysql_manager.log_system_event('ERROR', f\'Error al obtener predicciones: {e}\', module='admin', user_id=get_jwt_identity()[\id\'], ip_address=request.remote_addr) return jsonify({\message\': \'Error interno del servidor\'}), 500
```

```
@admin_bp.route('/logs', methods=['GET']) @admin_required def get_system_logs(): """Obtener todos los logs del sistema."""
try: logs = SystemLog.query.order_by(SystemLog.timestamp.desc()).all() logs_data = [{\id\': log.id, \level\': log.level, \message\': log.message, \module\': log.module, \timestamp\': log.timestamp.isoformat(), \user_id\': log.user_id, \username\': log.user.username if log.user else \'N/A\', \ip_address\': log.ip_address, \user_agent\': log.user_agent} for log in logs] mysql_manager.log_system_event('INFO', \'Acceso a logs del sistema por administrador\', module='admin', user_id=get_jwt_identity()[\id\'], ip_address=request.remote_addr) return jsonify(logs_data), 200 except Exception as e: logger.error(f\'Error al obtener logs: {e}\', exc_info=True) mysql_manager.log_system_event('ERROR', f\'Error al obtener logs: {e}\', module='admin', user_id=get_jwt_identity()[\id\'], ip_address=request.remote_addr) return jsonify({\message\': \'Error interno del servidor\'}), 500
```

```
@admin_bp.route('/stats', methods=['GET']) @admin_required def get_system_stats(): """Obtener estadísticas generales del sistema."""
try: stats = SystemStats.query.all() stats_data = {stat.stat_name: stat.stat_value for stat in stats} mysql_manager.log_system_event('INFO', \'Acceso a estadísticas del sistema por administrador\', module='admin', user_id=get_jwt_identity()[\id\'], ip_address=request.remote_addr) return jsonify(stats_data), 200 except Exception as e: logger.error(f\'Error al obtener estadísticas: {e}\', exc_info=True) mysql_manager.log_system_event('ERROR', f\'Error al obtener estadísticas: {e}\',
```

```
module='admin', user_id=get_jwt_identity()['id'],
ip_address=request.remote_addr) return jsonify({'message': 'Error interno del
servidor'}), 500

@admin_bp.route('/backup-db', methods=['POST']) @admin_required def
backup_database(): """Realizar una copia de seguridad de la base de datos.""" try:
    backup_file = mysql_manager.backup_database() if backup_file:
        mysql_manager.log_system_event('INFO', f'Backup de base de datos realizado:
{backup_file}', module='admin', user_id=get_jwt_identity()['id'],
ip_address=request.remote_addr) return jsonify({'message': 'Copia de seguridad
de la base de datos realizada exitosamente', 'file': backup_file}), 200 else:
        mysql_manager.log_system_event('ERROR', 'Fallo al realizar backup de base de
datos', module='admin', user_id=get_jwt_identity()['id'],
ip_address=request.remote_addr) return jsonify({'message': 'Fallo al realizar la
copia de seguridad de la base de datos'}), 500 except Exception as e:
    logger.error(f'Error en la ruta de backup: {e}', exc_info=True)
    mysql_manager.log_system_event('ERROR', f'Error en la ruta de backup: {e}', module='admin',
user_id=get_jwt_identity()['id'],
ip_address=request.remote_addr) return jsonify({'message': 'Error interno del
servidor al realizar backup'}), 500 ``
```

services/ai_service.py: Este es el módulo que encapsula toda la lógica relacionada con la inteligencia artificial. Contiene la clase `MangoAIService` que se encarga de cargar el modelo de TensorFlow, preprocesar las imágenes, realizar predicciones y proporcionar información detallada sobre las enfermedades. También incluye funciones de utilidad para la validación de archivos.

```

# v7.0/backend/services/ai_service.py

import tensorflow as tf
import numpy as np
from PIL import Image
import io
import logging
from config import settings as config

logger = logging.getLogger(__name__)

class MangoAIService:
    def __init__(self):
        self.model = None
        self.model_path = config.AI_MODEL_PATH
        self.image_size = 224 # Tamaño de entrada esperado por MobileNetV2
        self.disease_classes = config.DISEASE_CLASSES
        self.healthy_class = config.HEALTHY_CLASS_NAME

    def load_model(self):
        """Cargar el modelo de TensorFlow/Keras."""
        if self.model is None:
            try:
                self.model = tf.keras.models.load_model(self.model_path)
                logger.info(f"Modelo de IA cargado desde: {self.model_path}")
            except Exception as e:
                logger.error(f"Error cargando el modelo de IA desde {self.model_path}: {e}", exc_info=True)
                raise RuntimeError(f"No se pudo cargar el modelo de IA: {e}")
        return self.model

    def is_model_loaded(self):
        """Verifica si el modelo de IA está cargado."""
        return self.model is not None

    def preprocess_image(self, image_path):
        """Preprocesar la imagen para que sea compatible con el modelo de IA."""
        try:
            img = Image.open(image_path).resize((self.image_size,
                                                self.image_size))
            img_array = np.array(img) / 255.0 # Normalizar a [0, 1]
            img_array = np.expand_dims(img_array, axis=0) # Añadir dimensión de batch
            return img_array
        except Exception as e:
            logger.error(f"Error procesando imagen {image_path}: {e}", exc_info=True)
            raise ValueError(f"Error al preprocesar la imagen: {e}")

    def predict_disease(self, image_path):
        """Realizar la predicción de la enfermedad en la imagen."""
        try:
            model = self.load_model()
            if model is None:
                raise RuntimeError("Modelo de IA no cargado.")

            processed_image = self.preprocess_image(image_path)
            predictions = model.predict(processed_image)[0]

            # Obtener la clase predicha y su confianza
            predicted_class_index = np.argmax(predictions)
            predicted_class = self.disease_classes[predicted_class_index]
            confidence = float(predictions[predicted_class_index])
        
```

```

# Obtener todas las confianzas para cada clase
all_class_confidences = {
    self.disease_classes[i]: float(predictions[i])
    for i in range(len(self.disease_classes))
}

logger.info(f"Predicción: {predicted_class} con confianza
{confidence:.2f}")
return {
    "predicted_class": predicted_class,
    "confidence": confidence,
    "all_class_confidences": all_class_confidences
}
except Exception as e:
    logger.error(f"Error al predecir enfermedad para {image_path}: {e}" ,
exc_info=True)
    return None

def allowed_file(self, filename):
    """Verificar si el formato del archivo es permitido."""
    return "." in filename and \
        filename.rsplit(".", 1)[1].lower() in config.ALLOWED_IMAGE_FORMATS

def validate_image_content(self, file):
    """Validar el contenido de la imagen (tamaño, dimensiones)."""
    try:
        file.seek(0) # Asegurarse de leer desde el inicio del archivo
        img_bytes = file.read()
        image = Image.open(io.BytesIO(img_bytes))

        # Verificar tamaño del archivo
        if len(img_bytes) > config.MAX_CONTENT_LENGTH:
            raise ValueError(f"El archivo excede el tamaño máximo permitido de
{config.MAX_FILE_SIZE_MB}MB")

        # Verificar dimensiones mínimas
        if image.width < 50 or image.height < 50:
            raise ValueError("Imagen demasiado pequeña (mínimo 50x50
pixeles)")

        # Verificar dimensiones máximas
        if image.width > 5000 or image.height > 5000:
            raise ValueError("Imagen demasiado grande (máximo 5000x5000
pixeles)")

        return True

    except Exception as e:
        raise ValueError(f"Imagen inválida: {str(e)}")

def get_model_info(self):
    """Obtener información del modelo"""
    try:
        model = self.load_model()

        return {
            "model_name": "MobileNetV2 - Mango Disease Detection",
            "model_path": self.model_path,
            "input_size": f'\'{self.image_size}x{self.image_size}\',
            "classes": self.disease_classes,
            "total_classes": len(self.disease_classes),
            "healthy_class": self.healthy_class,
            "model_loaded": model is not None,
        }
    
```

```

        "supported_formats": config.ALLOWED_IMAGE_FORMATS,
        "max_file_size_mb": config.MAX_FILE_SIZE_MB
    }
except Exception as e:
    logger.error(f"Error obteniendo info del modelo: {e}")
    return None

def get_disease_info(self, disease_name):
    """Obtener información detallada de una enfermedad"""
    disease_info = {
        '\antracnosis': {
            '\name': '\Antracnosis',
            '\scientific_name': '\Colletotrichum gloeosporioides',
            '\type': '\Enfermedad fúngica',
            '\symptoms': '\Manchas oscuras circulares en hojas y frutos',
            '\treatment': '\Fungicidas a base de cobre, mejora de
ventilación',
            '\prevention': '\Evitar humedad excesiva, poda sanitaria'
        },
        '\chancro bacteriano': {
            '\name': '\Chancro Bacteriano',
            '\scientific_name': '\Xanthomonas citri',
            '\type': '\Enfermedad bacteriana',
            '\symptoms': '\Lesiones con halo amarillo en hojas',
            '\treatment': '\Bactericidas a base de cobre',
            '\prevention': '\Evitar heridas, desinfección de herramientas'
        },
        '\cortar gorgojo': {
            '\name': '\Daño por Gorgojo',
            '\scientific_name': '\Sternochetus mangiferae',
            '\type': '\Plaga de insectos',
            '\symptoms': '\Perforaciones y galerías en hojas',
            '\treatment': '\Insecticidas específicos, control biológico',
            '\prevention': '\Monitoreo regular, trampas'
        },
        '\die back': {
            '\name': '\Die Back',
            '\scientific_name': '\Lasiodiplodia theobromae',
            '\type': '\Enfermedad fúngica',
            '\symptoms': '\Muerte regresiva de ramas y hojas',
            '\treatment': '\Poda de partes afectadas, fungicidas',
            '\prevention': '\Evitar estrés hídrico, nutrición adecuada'
        },
        '\ball Midge': {
            '\name': '\Mosquito de la Bola',
            '\scientific_name': '\Procontarinia matteiana',
            '\type': '\Plaga de insectos',
            '\symptoms': '\Deformaciones en hojas jóvenes',
            '\treatment': '\Insecticidas sistémicos',
            '\prevention': '\Eliminación de restos vegetales'
        },
        '\saludable': {
            '\name': '\Planta Saludable',
            '\scientific_name': '\Mangifera indica',
            '\type': '\Estado normal',
            '\symptoms': '\Hojas verdes sin manchas ni deformaciones',
            '\treatment': '\Mantenimiento preventivo',
            '\prevention': '\Cuidados regulares, nutrición balanceada'
        },
        '\moho polvoriento': {
            '\name': '\Moho Polvoriento',
            '\scientific_name': '\Oidium mangiferae',
            '\type': '\Enfermedad fúngica',
            '\symptoms': '\Polvo blanco en superficie de hojas'
        }
    }

```

```

        \'treatment\': \'Fungicidas específicos para oidio\',  

        \'prevention\': \'Ventilación adecuada, evitar humedad\'  

    },  

    \'moho hollín\': {  

        \'name\': \'Moho Hollín\',  

        \'scientific_name\': \'Capnodium sp.\',  

        \'type\': \'Enfermedad fúngica\',  

        \'symptoms\': \'Capa negra tipo hollín en hojas\',  

        \'treatment\': \'Control de insectos, limpieza de hojas\',  

        \'prevention\': \'Control de pulgones y cochinillas\'  

    }  

}  
  

return disease_info.get(disease_name.lower(), {  

    \'name\': disease_name,  

    \'type\': \'Información no disponible\',  

    \'symptoms\': \'Consulte con un especialista\',  

    \'treatment\': \'Consulte con un especialista\',  

    \'prevention\': \'Consulte con un especialista\'  

})

```

ai_service = MangoAIService()

3.3. Frontend (React - Vite JSX)

El frontend del sistema está construido con React, utilizando Vite como herramienta de construcción para un desarrollo rápido y optimizado. La interfaz de usuario es responsive y permite a los usuarios interactuar con el backend para cargar imágenes y visualizar los resultados de las predicciones. La estructura de directorios del frontend es la siguiente:

```
v7.0/frontend/
├── public/
│   └── favicon.ico
└── src/
    ├── App.css
    ├── App.jsx
    ├── App_simple_working.jsx
    ├── assets/
    │   └── react.svg
    ├── components/
    │   ├── AdminDashboard.jsx
    │   ├── AgricultorDashboard.jsx
    │   ├── Dashboard.jsx
    │   ├── Login.jsx
    │   ├── OperatorDashboard.jsx
    │   ├── Register.jsx
    │   └── ui/ (... componentes de UI reutilizables)
    ├── contexts/
    │   └── AuthContext.jsx
    ├── hooks/
    │   └── use-mobile.js
    ├── index.css
    ├── lib/
    │   └── utils.js
    └── main.jsx
    └── components.json
    └── eslint.config.js
    └── index.html
    └── jsconfig.json
    └── package.json
    └── package-lock.json
    └── pnpm-lock.yaml
    └── vite.config.js
```

src/App.jsx : Este es el componente principal de la aplicación React. Se encarga de la lógica de enrutamiento y de mostrar el dashboard adecuado según el rol del usuario (administrador, operador, agricultor) y su estado de autenticación. Utiliza `AuthContext` para gestionar el estado de autenticación globalmente.

```

// v7.0/frontend/src/App.jsx

import React, { useState } from 'react';
import { AuthProvider, useAuth } from './contexts/AuthContext';
import Login from './components/Login';
import Register from './components/Register';
import AdminDashboard from './components/AdminDashboard';
import OperatorDashboard from './components/OperatorDashboard';
import AgricultorDashboard from './components/AgricultorDashboard';
import { Loader2 } from 'lucide-react';
import './App.css';

// Componente para manejar la autenticación y mostrar el dashboard correcto
const AppContent = () => {
  const { user, loading, isAuthenticated } = useAuth();
  const [showRegister, setShowRegister] = useState(false);

  // Mostrar loading mientras se verifica la autenticación
  if (loading) {
    return (
      <div className="min-h-screen flex items-center justify-center bg-gray-50">
        <div className="text-center">
          <Loader2 className="w-8 h-8 animate-spin mx-auto text-green-600" />
          <p className="mt-2 text-gray-600">Cargando...</p>
        </div>
      </div>
    );
  }

  // Si no está autenticado, mostrar login o registro
  if (!isAuthenticated) {
    if (showRegister) {
      return (
        <Register
          onSwitchToLogin={() => setShowRegister(false)}
          onRegisterSuccess={() => {
            // El usuario será redirigido automáticamente al dashboard
            // después del registro exitoso
          }}
        />
      );
    } else {
      return (
        <Login
          onSwitchToRegister={() => setShowRegister(true)}
          onLoginSuccess={() => {
            // El usuario será redirigido automáticamente al dashboard
            // después del login exitoso
          }}
        />
      );
    }
  }

  // Si está autenticado, mostrar el dashboard correspondiente según el rol
  const renderDashboard = () => {
    switch (user?.role) {
      case 'administrador':
        return <AdminDashboard />;
      case 'operador':
        return <OperatorDashboard />;
      case 'agricultor':
        return <AgricultorDashboard />;
    }
  }
}

```

```
default:
    // Por defecto, mostrar dashboard de agricultor
    return <AgricultorDashboard />;
}

};

return renderDashboard();
};

// Componente principal de la aplicación
function App() {
    return (
        <AuthProvider>
            <div className="App">
                <AppContent />
            </div>
        </AuthProvider>
    );
}

export default App;
```

`src/context/AuthContext.jsx`: Este contexto de React proporciona el estado de autenticación global a toda la aplicación. Maneja el inicio y cierre de sesión, y almacena la información del usuario y el token JWT. Utiliza `axios` para realizar las llamadas a la API del backend.

```

// v7.0/frontend/src/context/AuthContext.jsx

import React, { createContext, useContext, useState, useEffect } from 'react';
import axios from 'axios';

const AuthContext = createContext();

export const AuthProvider = ({ children }) => {
  const [user, setUser] = useState(null);
  const [isAuthenticated, setIsAuthenticated] = useState(false);
  const [loading, setLoading] = useState(true);

  const API_URL = import.meta.env.VITE_API_URL || 'http://localhost:5000/api';

  useEffect(() => {
    const loadUserFromLocalStorage = () => {
      try {
        const storedUser = localStorage.getItem('user');
        const storedToken = localStorage.getItem('token');
        if (storedUser && storedToken) {
          setUser(JSON.parse(storedUser));
          setIsAuthenticated(true);
          axios.defaults.headers.common['Authorization'] = `Bearer
${storedToken}`;
        }
      } catch (error) {
        console.error('Error loading user from local storage:', error);
        logout(); // Clear corrupted data
      } finally {
        setLoading(false);
      }
    };
    loadUserFromLocalStorage();
  }, []);

  const login = async (username, password) => {
    try {
      const response = await axios.post(`${API_URL}/auth/login`, { username, password });
      const { access_token, user: userData } = response.data;
      localStorage.setItem('token', access_token);
      localStorage.setItem('user', JSON.stringify(userData));
      setUser(userData);
      setIsAuthenticated(true);
      axios.defaults.headers.common['Authorization'] = `Bearer ${access_token}`;
      return { success: true };
    } catch (error) {
      console.error('Login failed:', error.response?.data || error.message);
      return { success: false, message: error.response?.data?.message || 'Error de inicio de sesión' };
    }
  };

  const register = async (username, email, password, role = 'agricultor') => {
    try {
      const response = await axios.post(`${API_URL}/auth/register`, { username, email, password, role });
      // Opcional: Iniciar sesión automáticamente después del registro
      // await login(username, password);
      return { success: true, message: response.data.message };
    } catch (error) {
      console.error('Registration failed:', error.response?.data || error.message);
    }
  };
}

```

```

        return { success: false, message: error.response?.data?.message || 'Error de
registro' };
    }
};

const logout = () => {
    localStorage.removeItem('token');
    localStorage.removeItem('user');
    setUser(null);
    setIsAuthenticated(false);
    delete axios.defaults.headers.common['Authorization'];
};

return (
    <AuthContext.Provider value={{ user, isAuthenticated, loading, login,
register, logout, API_URL }}>
        {children}
    </AuthContext.Provider>
);
};

export const useAuth = () => useContext(AuthContext);

```

src/components/ : Este directorio contiene los componentes de React que construyen la interfaz de usuario. Cada archivo `.jsx` representa un componente reutilizable o una vista principal de la aplicación.

- **Login.jsx** y **Register.jsx** : Componentes para el inicio de sesión y el registro de usuarios, respectivamente. Interactúan con las rutas de autenticación del backend.
- **AdminDashboard.jsx** , **OperatorDashboard.jsx** , **AgricultorDashboard.jsx** : Componentes que representan los paneles de control específicos para cada rol de usuario. Contienen la lógica y la UI para las funcionalidades permitidas a cada tipo de usuario.
- **ui/** : Contiene una colección de componentes de interfaz de usuario genéricos y reutilizables (por ejemplo, botones, entradas de texto, modales) que siguen un sistema de diseño consistente (probablemente Shadcn UI o similar, dado el `components.json` y las dependencias `@radix-ui`).

3.4. Base de Datos (MySQL)

La base de datos MySQL es el repositorio central de información para el sistema. Se utiliza para almacenar datos de usuarios, historial de predicciones, logs del sistema y estadísticas operativas. La interacción con la base de datos se realiza a través de SQLAlchemy ORM en el backend de Flask.

Esquema de la Base de Datos:

El esquema de la base de datos se define en `v7.0/backend/database/models.py` y consta de las siguientes tablas principales:

- **user** : Almacena la información de los usuarios del sistema.
 - `id` (PK, INT)
 - `username` (VARCHAR, UNIQUE)
 - `email` (VARCHAR, UNIQUE)
 - `password_hash` (VARCHAR)
 - `role` (VARCHAR) - `administrador` , `operador` , `agricultor`
 - `created_at` (DATETIME)
 - `updated_at` (DATETIME)
- **prediction_history** : Registra cada predicción de enfermedad realizada.
 - `id` (PK, INT)
 - `user_id` (FK a `user.id`, INT)
 - `image_path` (VARCHAR) - Nombre del archivo de la imagen analizada.
 - `predicted_class` (VARCHAR) - Clase de enfermedad predicha.
 - `confidence` (FLOAT) - Nivel de confianza de la predicción.
 - `prediction_date` (DATETIME)
 - `all_class_confidences` (TEXT) - Probabilidades para todas las clases (JSON string).
- **system_log** : Almacena los eventos y logs del sistema para auditoría y depuración.
 - `id` (PK, INT)
 - `level` (VARCHAR) - `INFO` , `WARNING` , `ERROR` , `CRITICAL`
 - `message` (TEXT)
 - `module` (VARCHAR) - Módulo de origen del log (e.g., `auth` , `disease` , `admin`).
 - `timestamp` (DATETIME)
 - `user_id` (FK a `user.id`, INT, NULLABLE)
 - `ip_address` (VARCHAR)
 - `user_agent` (VARCHAR)

- **system_stats** : Guarda estadísticas operativas clave del sistema.
 - `id` (PK, INT)
 - `stat_name` (VARCHAR, UNIQUE) - Nombre de la estadística (e.g., `total_users`, `total_predictions`).
 - `stat_value` (TEXT) - Valor de la estadística.
 - `last_updated` (DATETIME)

3.5. Inteligencia Artificial (TensorFlow)

El componente de IA es fundamental para la funcionalidad principal del sistema: la detección de enfermedades en hojas de mango. Se utiliza un modelo de aprendizaje profundo pre-entrenado con TensorFlow, específicamente la arquitectura MobileNetV2, conocida por su eficiencia y buen rendimiento en tareas de clasificación de imágenes. El modelo `mango_disease_model_mobilenetv2.h5` se encuentra en el directorio `v7.0/backend/models/`.

Funcionamiento del Modelo de IA:

1. **Carga del Modelo:** El `ai_service.py` carga el modelo `.h5` en memoria al iniciar la aplicación Flask. Esto asegura que el modelo esté listo para realizar inferencias rápidamente sin recargarse en cada solicitud.

```
```python
```

---

## Fragmento de services/ai\_service.py

```
def load_model(self): """Cargar el modelo de TensorFlow/Keras.""" if self.model is None: try: self.model = tf.keras.models.load_model(self.model_path) logger.info(f"Modelo de IA cargado desde: {self.model_path}") except Exception as e: logger.error(f"Error cargando el modelo de IA desde {self.model_path}: {e} , exc_info=True) raise RuntimeError(f"No se pudo cargar el modelo de IA: {e}") return self.model```
```

2. **Preprocesamiento de Imágenes:** Antes de que una imagen pueda ser alimentada al modelo, debe ser preprocesada para que coincida con el formato de entrada

esperado por MobileNetV2 (imágenes de 224x224 píxeles, normalizadas a un rango de [0, 1]).

```
```python
```

Fragmento de services/ai_service.py

```
def preprocess_image(self, image_path): """Preprocesar la imagen para que sea compatible con el modelo de IA.""" try: img = Image.open(image_path).resize((self.image_size, self.image_size)) img_array = np.array(img) / 255.0 # Normalizar a [0, 1] img_array = np.expand_dims(img_array, axis=0) # Añadir dimensión de batch return img_array except Exception as e: logger.error(f"Error preprocesando imagen {image_path}: {e}" , exc_info=True) raise ValueError(f"Error al preprocesar la imagen: {e}") ````
```

3. Predicción: Una vez preprocesada, la imagen se pasa al modelo para la inferencia. El modelo devuelve un array de probabilidades, donde cada valor corresponde a la confianza de que la imagen pertenezca a una de las ocho clases de enfermedades o a la clase

saludable.

```

```python
Fragmento de services/ai_service.py
def predict_disease(self, image_path):
 """Realizar la predicción de la enfermedad en la imagen."""
 try:
 model = self.load_model()
 if model is None:
 raise RuntimeError("Modelo de IA no cargado.")

 processed_image = self.preprocess_image(image_path)
 predictions = model.predict(processed_image)[0]

 # Obtener la clase predicha y su confianza
 predicted_class_index = np.argmax(predictions)
 predicted_class = self.disease_classes[predicted_class_index]
 confidence = float(predictions[predicted_class_index])

 # Obtener todas las confianzas para cada clase
 all_class_confidences = {
 self.disease_classes[i]: float(predictions[i])
 for i in range(len(self.disease_classes))
 }

 logger.info(f"Predicción: {predicted_class} con confianza {confidence:.2f}")
 except Exception as e:
 logger.error(f"Error al predecir enfermedad para {image_path}: {e}", exc_info=True)
 return {
 "predicted_class": predicted_class,
 "confidence": confidence,
 "all_class_confidences": all_class_confidences
 }
```

```

1. Información de Clases de Enfermedades: El servicio de IA también proporciona información detallada sobre cada una de las ocho enfermedades que puede detectar, así como para el estado saludable. Esta información incluye el nombre común, nombre científico, tipo (fúngica, bacteriana, plaga), síntomas, tratamiento y prevención. Esta base de conocimientos es crucial para ofrecer recomendaciones útiles a los usuarios.

```

```python

```

## Fragmento de services/ai\_service.py

---

```

def get_disease_info(self, disease_name):
 """Obtener información detallada de una enfermedad"""
 disease_info = {
 'antracnosis': {
 'name': 'Antracnosis',
 'scientific_name': 'Colletotrichum gloeosporioides',
 'type': 'Enfermedad fúngica'
 }
 }

```

'symptoms': 'Manchas oscuras circulares en hojas y frutos', 'treatment': 'Fungicidas a base de cobre, mejora de ventilación', 'prevention': 'Evitar humedad excesiva, poda sanitaria' }, 'chancro bacteriano': { 'name': 'Chancro Bacteriano', 'scientific\_name': 'Xanthomonas citri', 'type': 'Enfermedad bacteriana', 'symptoms': 'Lesiones con halo amarillo en hojas', 'treatment': 'Bactericidas a base de cobre', 'prevention': 'Evitar heridas, desinfección de herramientas' }, 'cortar gorgojo': { 'name': 'Daño por Gorgojo', 'scientific\_name': 'Sternochetus mangiferae', 'type': 'Plaga de insectos', 'symptoms': 'Perforaciones y galerías en hojas', 'treatment': 'Insecticidas específicos, control biológico', 'prevention': 'Monitoreo regular, trampas' }, 'die back': { 'name': 'Die Back', 'scientific\_name': 'Lasiodiplodiella theobromae', 'type': 'Enfermedad fúngica', 'symptoms': 'Muerte regresiva de ramas y hojas', 'treatment': 'Poda de partes afectadas, fungicidas', 'prevention': 'Evitar estrés hídrico, nutrición adecuada' }, 'ball Midge': { 'name': 'Mosquito de la Bola', 'scientific\_name': 'Procontarinia matteiana', 'type': 'Plaga de insectos', 'symptoms': 'Deformaciones en hojas jóvenes', 'treatment': 'Insecticidas sistémicos', 'prevention': 'Eliminación de restos vegetales' }, 'saludable': { 'name': 'Planta Saludable', 'scientific\_name': 'Mangifera indica', 'type': 'Estado normal', 'symptoms': 'Hojas verdes sin manchas ni deformaciones', 'treatment': 'Mantenimiento preventivo', 'prevention': 'Cuidados regulares, nutrición balanceada' }, 'moho polvoriento': { 'name': 'Moho Polvoriento', 'scientific\_name': 'Oidium mangiferae', 'type': 'Enfermedad fúngica', 'symptoms': 'Polvo blanco en superficie de hojas', 'treatment': 'Fungicidas específicos para oídio', 'prevention': 'Ventilación adecuada, evitar humedad' }, 'moho hollín': { 'name': 'Moho Hollín', 'scientific\_name': 'Capnodium sp.', 'type': 'Enfermedad fúngica', 'symptoms': 'Capa negra tipo hollín en hojas', 'treatment': 'Control de insectos, limpieza de hojas', 'prevention': 'Control de pulgones y cochinillas' } }

```
return disease_info.get(disease_name.lower(), {
 'name': disease_name,
 'type': 'Información no disponible',
 'symptoms': 'Consulte con un especialista',
 'treatment': 'Consulte con un especialista',
 'prevention': 'Consulte con un especialista'
})
```

```

Esta integración permite que el sistema no solo identifique la enfermedad, sino que también proporcione información valiosa para el manejo y control de las mismas, convirtiéndose en una herramienta de apoyo a la decisión para los usuarios finales.

4. Fundamento Científico: Enfermedades de la Hoja de Mango

El sistema de detección se basa en la identificación de ocho enfermedades comunes que afectan las hojas del mango, además de diferenciar una hoja saludable. A continuación, se presenta una descripción científica de cada una, incluyendo sus agentes causales, síntomas característicos, tratamientos y medidas de prevención. Esta información es crucial para la interpretación de los resultados del modelo de IA y para la toma de decisiones agronómicas.

4.1. Antracnosis

- **Nombre Científico del Agente Causal:** *Colletotrichum gloeosporioides*
- **Tipo:** Enfermedad fúngica.
- **Síntomas:** Se manifiesta como manchas oscuras, irregulares y hundidas en las hojas, que pueden coalesce y causar la muerte del tejido. En frutos, aparecen lesiones circulares negras que pueden llevar a la pudrición. También afecta flores y ramas jóvenes, causando tizón y muerte regresiva.
- **Tratamiento:** Aplicación de fungicidas a base de cobre o sistémicos. La poda sanitaria de ramas infectadas es fundamental. Control de la humedad en el dosel del árbol.
- **Prevención:** Uso de variedades resistentes, manejo adecuado de la densidad de siembra para mejorar la ventilación, eliminación de restos de cultivo infectados y aplicación preventiva de fungicidas durante períodos de alta humedad.

4.2. Chancro Bacteriano

- **Nombre Científico del Agente Causal:** *Xanthomonas citri* pv. *mangiferaeindicae*
- **Tipo:** Enfermedad bacteriana.
- **Síntomas:** Lesiones pequeñas, angulares y empapadas de agua en las hojas, que con el tiempo se vuelven necróticas con un halo amarillo distintivo. Pueden aparecer chancros en ramas y frutos, causando deformaciones y exudación gomosa.
- **Tratamiento:** No existen tratamientos curativos efectivos una vez establecida la infección. Los bactericidas a base de cobre pueden ayudar a reducir la propagación. La eliminación de material vegetal infectado es crucial.

- **Prevención:** Uso de material de siembra sano, evitar heridas en las plantas, desinfección de herramientas de poda y control de insectos vectores. La exclusión es la mejor estrategia.

4.3. Daño por Gorgojo (Cortar Gorgojo)

- **Nombre Científico del Agente Causal:** *Sternochetus mangiferae* (Gorgojo de la semilla del mango) u otras especies de gorgojos que afectan las hojas.
- **Tipo:** Plaga de insectos (Coleoptera: Curculionidae).
- **Síntomas:** Las hojas presentan perforaciones irregulares, galerías o cortes en los márgenes, resultado de la alimentación de los adultos o larvas. En el caso del gorgojo de la semilla, el daño principal es interno en el fruto, pero los adultos pueden alimentarse de las hojas.
- **Tratamiento:** Aplicación de insecticidas específicos en etapas tempranas de la infestación. Control biológico con enemigos naturales. Recolección y destrucción de frutos caídos o infestados.
- **Prevención:** Monitoreo regular de la población de gorgojos, uso de trampas de feromonas, prácticas culturales como la labranza para exponer pupas y la eliminación de malezas que puedan servir de refugio.

4.4. Die Back (Muerte Regresiva)

- **Nombre Científico del Agente Causal:** *Lasiodiplodia theobromae*
- **Tipo:** Enfermedad fúngica.
- **Síntomas:** Caracterizada por la muerte progresiva de las ramas desde la punta hacia la base. Las hojas se marchitan, se vuelven marrones y permanecen adheridas a las ramas muertas. Puede haber exudación gomosa en los puntos de infección.
- **Tratamiento:** Poda severa de las ramas afectadas, cortando varios centímetros por debajo del tejido visiblemente enfermo. Aplicación de fungicidas protectores en los cortes y en el árbol. Mejora de la nutrición y el riego para fortalecer la planta.
- **Prevención:** Evitar el estrés hídrico y nutricional, proteger los árboles de heridas mecánicas, y realizar podas sanitarias regulares para eliminar material muerto o débil.

4.5. Mosquito de la Bola (Ball Midge)

- **Nombre Científico del Agente Causal:** *Procontarinia matteiana* (Díptera: Cecidomyiidae).
 - **Tipo:** Plaga de insectos.
 - **Síntomas:** Las larvas de este mosquito inducen la formación de agallas esféricas o globulares en las hojas jóvenes, que pueden deformarlas y reducir la capacidad fotosintética. Las agallas son inicialmente verdes y luego se vuelven marrones y duras. *
- Tratamiento:** En infestaciones severas, se pueden usar insecticidas sistémicos. La eliminación manual de las hojas infestadas y su destrucción puede ser efectiva en árboles pequeños. * **Prevención:** Monitoreo de brotes nuevos, eliminación de restos vegetales y hojas caídas que puedan albergar pupas, y fomento de enemigos naturales.

4.6. Planta Saludable

- **Nombre Científico:** *Mangifera indica*
- **Tipo:** Estado normal.
- **Síntomas:** Hojas de color verde intenso y uniforme, sin manchas, deformaciones, perforaciones, decoloraciones o crecimientos anormales. La superficie de la hoja es lisa y brillante, y la planta muestra un crecimiento vigoroso y equilibrado.
- **Tratamiento:** Mantenimiento preventivo, que incluye riego adecuado, fertilización balanceada, poda de formación y saneamiento, y control general de plagas y enfermedades para evitar su aparición.
- **Prevención:** Implementación de buenas prácticas agrícolas, monitoreo constante del estado de salud de la planta y del suelo, y uso de variedades adaptadas a las condiciones locales.

4.7. Moho Polvoriento

- **Nombre Científico del Agente Causal:** *Oidium mangiferae*
- **Tipo:** Enfermedad fungica.
- **Síntomas:** Se caracteriza por la aparición de un crecimiento blanco y polvoriento en la superficie de las hojas jóvenes, brotes, flores y frutos. Las hojas afectadas pueden enrollarse, distorsionarse y caer prematuramente. En infestaciones severas, puede cubrir completamente las superficies, impidiendo la fotosíntesis.

- **Tratamiento:** Aplicación de fungicidas específicos para oídio, como los basados en azufre o fungicidas sistémicos. Es importante la cobertura total de la planta.
- **Prevención:** Mejorar la circulación del aire en el dosel del árbol mediante podas adecuadas, evitar el exceso de humedad, y realizar aplicaciones preventivas de fungicidas en condiciones favorables para el desarrollo del hongo.

4.8. Moho Hollín

- **Nombre Científico del Agente Causal:** *Capnodium sp.* (y otros géneros de hongos saprófitos).
- **Tipo:** Enfermedad fúngica (saprófita, no parasitaria directamente de la planta).
- **Síntomas:** Se presenta como una capa negra, similar al hollín, que cubre la superficie de las hojas, ramas y frutos. Este moho crece sobre la melaza excretada por insectos chupadores como pulgones, cochinillas y moscas blancas. Aunque no parasita directamente la planta, la capa negra reduce la fotosíntesis y el intercambio gaseoso, afectando el vigor de la planta y la calidad de los frutos.
- **Tratamiento:** El tratamiento principal se enfoca en el control de los insectos chupadores que producen la melaza. Una vez controlada la plaga, el moho hollín puede lavarse con agua a presión o con soluciones jabonosas suaves. En casos severos, se pueden usar fungicidas.
- **Prevención:** Control efectivo de las poblaciones de insectos chupadores mediante insecticidas, control biológico o trampas. Mantener la higiene del huerto y evitar el exceso de humedad.

5. Ilustraciones Visuales

Esta sección presenta una colección de imágenes que ilustran las diferentes condiciones de las hojas de mango, incluyendo el estado saludable y las diversas enfermedades que el sistema es capaz de detectar. Estas imágenes sirven como referencia visual para comprender mejor los síntomas descritos en la sección anterior y para apreciar la complejidad de la tarea de clasificación que realiza el modelo de IA.

5.1. Hojas de Mango Saludables

Las hojas de mango saludables se caracterizan por su color verde vibrante, textura lisa y ausencia de manchas, deformaciones o signos de plagas. Reflejan un estado óptimo de salud de la planta y una adecuada nutrición.



Figura 5.1.1: Hojas de mango saludables, mostrando un color verde uniforme y sin imperfecciones.



Figura 5.1.2: Primer plano de hojas de mango sanas, destacando su superficie lisa y venación clara.

5.2. Antracnosis

La antracnosis se manifiesta con manchas oscuras y hundidas, a menudo con anillos concéntricos, que pueden expandirse y causar la necrosis de grandes áreas de la hoja.



Figura 5.2.1: Manchas circulares oscuras y hundidas características de la antracnosis en una hoja de mango.

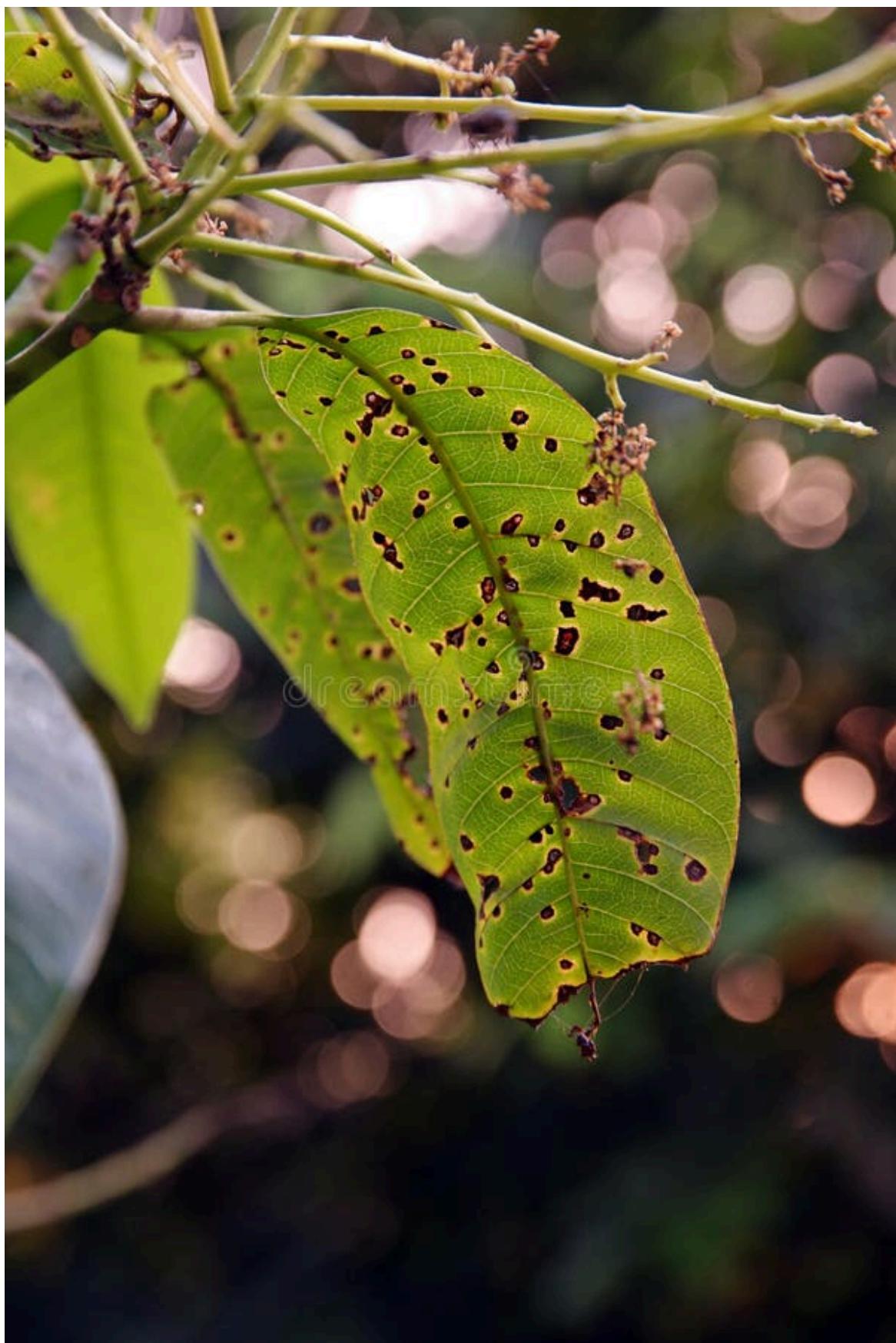


Figura 5.2.2: Antracnosis avanzada, mostrando la coalescencia de lesiones y la necrosis del tejido foliar.

5.3. Chancro Bacteriano

El chancro bacteriano se identifica por lesiones angulares, empapadas de agua, que se tornan necróticas y están rodeadas por un halo clorótico (amarillo).



Figura 5.3.1: Lesiones angulares con halo amarillo, indicativas de chancre bacteriano en hoja de mango.



Figura 5.3.2: Múltiples lesiones de chancre bacteriano en una hoja, con necrosis central y halos distintivos.

5.4. Daño por Gorgojo

El daño por gorgojo se evidencia por perforaciones irregulares, cortes o galerías en la superficie de la hoja, resultado de la actividad alimenticia del insecto.



Figura 5.4.1: Perforaciones y mordeduras en la hoja de mango causadas por gorgojos.



Figura 5.4.2: Daño severo por gorgojo, con grandes áreas de tejido foliar consumido.

5.5. Die Back

El

Die Back (muerte regresiva) se caracteriza por el marchitamiento y oscurecimiento de las hojas y ramas, progresando desde la punta hacia la base.



Figura 5.5.1: Ramas y hojas de mango mostrando signos de muerte regresiva (Die Back).



Figura 5.5.2: Hojas secas y marchitas, un síntoma común de la enfermedad Die Back.

5.6. Mosquito de la Bola (Ball Midge)

El mosquito de la bola provoca la formación de agallas esféricas o globulares en las hojas jóvenes, que pueden ser de diferentes tamaños y colores.



Figura 5.6.1: Agallas características causadas por el mosquito de la bola en la superficie de una hoja de mango.



shutterstock®

IMAGE ID: 1738840637
www.shutterstock.com

Figura 5.6.2: Múltiples agallas esféricas en una hoja, indicando una infestación de mosquito de la bola.

5.7. Moho Polvoriento

El moho polvoriento se presenta como una capa blanca y pulverulenta en la superficie de las hojas, que puede cubrir grandes áreas y afectar la fotosíntesis.



Figura 5.7.1: Crecimiento blanco y polvoriento en la superficie de una hoja de mango, síntoma de oídio.



Figura 5.7.2: Hojas de mango severamente afectadas por moho polvoriento, con deformación y enrollamiento.

5.8. Moho Hollín

El moho hollín se manifiesta como una capa negra y pegajosa que cubre las hojas, resultado de la melaza excretada por insectos chupadores. Esta capa reduce la capacidad fotosintética de la planta.



Figura 5.8.1: Capa negra similar al hollín cubriendo la superficie de las hojas de mango.





Figura 5.8.2: Acumulación de moho hollín en hojas, afectando la apariencia y la salud de la planta.

6. Conclusiones y Futuras Mejoras

El Sistema Web Inteligente para la Detección de Enfermedades en la Hoja de Mango representa un avance significativo en la aplicación de la inteligencia artificial para la agricultura. Al integrar un backend robusto con Flask, un frontend intuitivo con React y un modelo de IA basado en TensorFlow, el sistema ofrece una herramienta eficiente y accesible para la detección temprana de enfermedades en el cultivo de mango. La capacidad de identificar ocho tipos de enfermedades comunes, además del estado saludable, proporciona a los agricultores y agrónomos información crucial para la toma de decisiones informadas, lo que puede conducir a una reducción en el uso de pesticidas, una mejora en la productividad y una mayor sostenibilidad agrícola.

La arquitectura modular del sistema, con una clara separación entre el frontend, el backend y el servicio de IA, facilita su mantenimiento, escalabilidad y futuras expansiones. La utilización de tecnologías ampliamente adoptadas como Python, Flask, React, MySQL y TensorFlow asegura una base sólida y una comunidad de soporte activa.

Futuras Mejoras:

- **Expansión del Modelo de IA:** Integrar modelos más avanzados o entrenar el modelo actual con un conjunto de datos más amplio y diverso para mejorar la precisión y la capacidad de detección de nuevas enfermedades o variaciones de las existentes.
- **Detección en Tiempo Real:** Explorar la posibilidad de integrar el sistema con dispositivos móviles o cámaras en campo para permitir la detección de enfermedades en tiempo real, lo que agilizaría aún más la respuesta.
- **Recomendaciones Personalizadas:** Desarrollar un módulo que, basándose en la predicción de la enfermedad y la ubicación geográfica, ofrezca recomendaciones de tratamiento y prevención más específicas y adaptadas a las condiciones locales.

- **Integración con Sistemas de Gestión Agrícola:** Conectar el sistema con plataformas existentes de gestión agrícola para una visión más holística de la salud del cultivo y la planificación de actividades.
- **Interfaz de Usuario Mejorada:** Continuar optimizando la experiencia del usuario, incluyendo funcionalidades como el seguimiento del progreso de las enfermedades, notificaciones personalizadas y reportes detallados.
- **Análisis de Tendencias:** Implementar herramientas para analizar tendencias de enfermedades a lo largo del tiempo y en diferentes ubicaciones, lo que podría ayudar en la investigación y la formulación de políticas agrícolas.

En resumen, este sistema no solo aborda una necesidad crítica en la producción de mango, sino que también sienta las bases para futuras innovaciones en la agricultura de precisión, demostrando el potencial transformador de la inteligencia artificial en el sector agrícola.

7. Referencias

7.1. Referencias de Imágenes

- **Hojas de Mango Saludables:**
 - Figura 5.1.1: [Mejor con Salud](#)
 - Figura 5.1.2: [Pixabay](#)
- **Antracnosis:**
 - Figura 5.2.1: [Dreamstime](#)
 - Figura 5.2.2: [Dreamstime](#)
- **Chancro Bacteriano:**
 - Figura 5.3.1: [Universidad Nacional de Asunción](#)
 - Figura 5.3.2: [Dreamstime](#)
- **Daño por Gorgojo:**
 - Figura 5.4.1: [Plantix](#)
 - Figura 5.4.2: [Plantix](#)

- **Die Back:**

- Figura 5.5.1: [Plantix](#)
- Figura 5.5.2: [Plantix](#)

- **Mosquito de la Bola (Ball Midge):**

- Figura 5.6.1: [Greenlife Crop Protection Africa](#)
- Figura 5.6.2: [Shutterstock](#)

- **Moho Polvoriento:**

- Figura 5.7.1: [Reddit](#)
- Figura 5.7.2: [Plantix](#)

- **Moho Hollín:**

- Figura 5.8.1: [Reddit](#)
- Figura 5.8.2: [PictureThis](#)