CAPÍTULO 7

BACKEND PYTHON FLASK: API RESTFUL, ARQUITETURA DE ROTAS, PERSISTÊNCIA MYSQL E TESTES DE INTEGRAÇÃO NA IRRIGAÇÃO IoT

1. INTRODUÇÃO AO BACKEND PARA O LEITOR AGRÔNOMO: O MOTOR CENTRAL DA FAZENDA DIGITAL

Para um engenheiro agrônomo, produtor rural ou gestor de lavoura, a compreensão da infraestrutura computacional que sustenta a agricultura digital torna-se clara ao utilizar a analogia de uma **Casa de Máquinas Central** em um projeto de irrigação por gotejamento ou pivô central.

Em uma fazenda automatizada, a lavoura pode possuir quilômetros de tubulações hidráulicas, dezenas de válvulas solenoides instaladas nos talhões e painéis solares no campo. No entanto, nenhum setor irrigará uma única gota de água se a casa de máquinas central — onde ficam alocados a motobomba principal, os dosadores de fertirrigação, os filtros de disco, os quadros elétricos de comando e os manômetros de pressão — não estiver funcionando perfeitamente e enviando o fluxo de água correto para o setor desejado.

Na engenharia de software, o **Backend** é exatamente a "casa de máquinas central" do sistema de informação (SOMMERVILLE, 2016). Enquanto o *Frontend* (Capítulo 6) representa o painel visual com o qual o operador humano interage e os microcontroladores ESP32 (Capítulo 5) representam os acionadores elétricos no campo, o Backend é a inteligência invisível que roda em um servidor remoto, processando as regras de negócio, gerenciando as conexões com o banco de dados e garantindo que o comando enviado pelo agrônomo seja executado com total segurança (MARTIN, 2017).

Na aplicação hipotética desenvolvida no diretório /sistema/backend-irriga, o servidor Backend foi construído utilizando a linguagem de programação **Python** e o framework web **Flask**. Este capítulo tem por objetivo explicar detalhadamente cada componente desse backend, apresentando seus códigos-fonte, sua arquitetura de rotas, seu banco de dados MySQL e as ferramentas de teste de integração.

2. O FRAMEWORK PYTHON FLASK: O CHASSI ÁGIL DA AUTOMAÇÃO

No ecossistema de desenvolvimento de software em linguagem Python, existem diversos frameworks para a criação de aplicações web. O **Flask** é classificado pela comunidade técnica como um *micro-framework*.

Para o leitor agrônomo, o conceito de *micro-framework* pode ser compreendido mediante a comparação entre dois tipos de maquinários agrícolas:

- **Frameworks Pesados (Estilo Colheitadeira Rígida):** Frameworks como o Django fornecem uma estrutura gigante e pré-configurada "tudo-em-um". São excelentes para grandes portais corporativos, mas trazem um peso computacional elevado e pouca flexibilidade para customizações de baixo nível necessárias na

Internet das Coisas.

- **Micro-Framework Flask (Estilo Trator Utilitário Modular):** O Flask fornece apenas o essencial — o chassi mecânico ágil com motor de alto rendimento, o sistema de roteamento de mensagens HTTP e o gerenciamento de requisições. Cabe aos desenvolvedores acoplar apenas os implementos estritamente necessários (como a biblioteca SQLAlchemy para banco de dados e o repositório de rotas Blueprints).

Devido à sua extrema leveza, rapidez de resposta e eficiência no processamento de requisições concorrentes, o Flask é a escolha ideal para atuar como o cérebro central de plataformas de irrigação inteligente IoT.

3. CONCEITUAÇÃO DE ROTAS HTTP E ENDPOINTS USANDO O DOMÍNIO www.meusitedeirrigacao.com

Para que a interface web em Angular, o aplicativo de celular do agrônomo ou o micro-controlador ESP32 consigam conversar com o servidor Flask, eles utilizam **Rotas HTTP** (também chamadas de *Endpoints*).

Uma Rota HTTP funciona como um **endereço postal digital** ou uma **válvula seletora de distribuição hidráulica**. Cada rota possui um nome de caminho específico e aceita um ou mais métodos HTTP (conforme apresentado no Capítulo 4), definindo exatamente qual ação o servidor deve executar.

Para ilustrar o funcionamento prático das rotas no sistema hipotético, consideremos que a plataforma de irrigação da fazenda está publicada na internet sob o domínio corporativo oficial:

http://www.meusitedeirrigacao.com

A seguir, apresentam-se exemplos práticos de como cada método HTTP é utilizado no domínio www.meusitedeirrigacao.com para controlar a infraestrutura de irrigação.

3.1. Método GET: Consulta e Leitura de Dados

O método *GET* é utilizado quando um cliente (navegador ou ESP32) deseja ler informações armazenadas no servidor, sem alterar nenhuma configuração no banco de dados:

- **Consulta do Estado Geral pelo ESP32:**

GET http://www.meusitedeirrigacao.com/estado-geral

O servidor processa a consulta, verifica os agendamentos ativos para o horário atual

e retorna o documento JSON informando se a bomba e as quatro solenoides devem estar ligadas.

- **Listagem de Agendamentos pelo Frontend Web:**

GET http://www.meusitedeirrigacao.com/agendamentos

O servidor realiza uma busca na tabela MySQL e devolve a lista completa de todas as regas programadas na fazenda.

3.2. Método POST: Criação e Disparo de Comandos

O método *POST* é utilizado para submeter novos dados ao servidor, criando registros no banco de dados ou alterando manualmente o estado elétrico de um atuador:

- **Cadastro de Novo Agendamento de Irrigação:**

POST http://www.meusitedeirrigacao.com/agendamentos

O formulário web do agrônomo envia no corpo da requisição os dados da nova rega (data, horário de início, duração e solenoide selecionada). O servidor Flask valida as informações e insere uma nova linha na tabela do MySQL.

- **Acionamento Manual de Solenoide de Campo:**

POST http://www.meusitedeirrigacao.com/solenoide?id=1&ativo=true

O operador força a abertura imediata da solenoide 1 (Setor Melão), atualizando o banco de dados.

3.3. Método PUT: Atualização Completa de Registros

O método *PUT* é empregado para modificar as informações de um recurso já existente no sistema:

- **Alteração de Horário de um Agendamento Existente:**

PUT http://www.meusitedeirrigacao.com/agendamentos/12

O agrônomo decide prorrogar a rega do agendamento de identificador ID 12. O

cliente envia os novos horários e o servidor executa uma instrução SQL do tipo *UPDATE* na linha correspondente do banco MySQL.

3.4. Método DELETE: Remoção e Cancelamento

O método *DELETE* é utilizado para excluir permanentemente um registro do banco de dados:

- **Cancelamento de uma Rega Programada:**

DELETE http://www.meusitedeirrigacao.com/agendamentos/12

O operador cancela a rega ID 12 porque começou a chover na propriedade. O servidor recebe a instrução e executa o comando *SQL DELETE FROM* agendamentos *WHERE id = 12.*

4. ANÁLISE ESTRUTURAL E CÓDIGO-FONTE DO ARQUIVO PRINCIPAL *app.py*

O arquivo *app.py* (localizado em /sistema/backend-irriga/app.py) é o ponto de entrada principal do servidor Flask. Ele é responsável por inicializar a aplicação web, configurar a conexão com o banco de dados MySQL, registrar os módulos de rotas e iniciar o servidor de escuta.

A Listagem 1 apresenta o código-fonte exato do arquivo *app.py.*

Listing 4: Código-fonte do arquivo principal app.py do servidor backend Flask.

```
import os
import sys
from flask import Flask, request, jsonify
from flask_sqlalchemy import SQLAlchemy
from sqlalchemy import text
from flask_cors import CORS
from routes.bomba_routes import bomba_routes
from routes.agendamento_routes import agendamento_routes
import urllib.parse

from db import db
app = Flask(__name__)
app.secret_key = 'sua_chave_secreta_aqui'

CORS(app, expose_headers=["X-Filename", "Content-Disposition"])

password = "Irriga@2026"
encoded_password = urllib.parse.quote_plus(password)

db_host = os.environ.get('DB_HOST', 'localhost')
app.config['SQLALCHEMY_DATABASE_URI'] = (
    f'mysql+pymysql://sergiosouzanovak_sergiosouzanovak:{encoded_password}'
    f'@{db_host}/sergiosouzanovak_sistema_irrigacao'
)
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
app.config['SQLALCHEMY_ENGINE_OPTIONS'] = {
    'pool_recycle': 280,
    'pool_pre_ping': True,
}
db.init_app(app)
app.register_blueprint(bomba_routes)
app.register_blueprint(agendamento_routes)

if __name__ == "__main__":
    app.run(debug=True, host='0.0.0.0', port=5000)
```

4.1. Análise Linha a Linha das Configurações do Servidor

A análise técnica do arquivo *app.py* revela decisões cruciais de arquitetura de software:

1. **Importação de Módulos e Bibliotecas (Linhas 1 a 12):** O script importa as classes do Flask, a extensão *SQLAlchemy* para mapeamento de banco de dados, o manipulador de segurança de rede *CORS* e os dois conjuntos de rotas contidos na pasta *routes/.*
2. **Tratamento de Senha de Banco de Dados com Caracteres Especiais (Linhas 19 e 20):** A senha do banco de dados MySQL contém o caractere especial @ *("Irriga@2026")*. Como a URL de conexão com o banco utiliza o caractere @ para separar a senha do endereço IP do servidor, o código utiliza a função *urllib.parse.quote_plus()* para codificar o caractere em formato percentual *(%40*). Isso impede que o Python confunda a senha com o endereço de rede do banco MySQL.
3. **Configuração da Conexão MySQL (Linhas 23 a 26):** A variável *SQLALCHEMY_DATABASE_URI* monta a string de conexão utilizando o driver *pymysql*, conectando-se ao banco de dados relacional *sergiosouzanovak_sistema_irrigacao.*
4. **Gestão de Pool de Conexões e Reconexão Autônoma (Linhas 28 a 31):** Em servidores de hospedagem web rurais, as conexões inativas com o banco de dados MySQL podem ser encerradas por limite de tempo (*timeout*). A configuração *'pool_recycle': 280* força o Flask a renovar as conexões a cada 280 segundos. Além disso, o parâmetro *'pool_pre_ping*': True ativa a sondagem prévia: antes de exe-cutar qualquer consulta SQL, o Flask testa a conexão; se o MySQL tiver caído, o Flask reconecta automaticamente de forma transparente.
5. **Registros de Blueprints de Rotas (Linhas 33 e 34):** A instrução *app.register_blueprint()* acopla os módulos independentes de rotas de atuadores (*bomba_routes*) e de agendamentos (*agendamento_routes*) à aplicação principal, garantindo uma estrutura modular de código.

5. A LÓGICA DE SINCRONIZAÇÃO AUTOMÁTICA DE STATUS: A FUNÇÃO *sincronizar_status()*

Um dos pontos mais geniais e importantes de todo o backend está localizado no arquivo *bomba_routes.py:* a função *sincronizar_status()* (linhas 8 a 54 da Listagem 2).

Para o leitor agrônomo, entender essa função é compreender a **automação em tempo real**. O servidor não depende que uma pessoa clique em um botão no momento exato em que a irrigação deve começar. A função *sincronizar_status()* atua como um relógio automático inteligente que avalia a hora atual do dia contra a tabela de agendamentos salvos.

5.1. O Algoritmo de Decisão de Acionamento Hidráulico

Sempre que o ESP32 ou o painel web realizam uma consulta ao servidor, a função *sincronizar_status()* executa as seguintes etapas:

1. **Leitura de Todos os Solenoides Cadastrados:** O servidor consulta a tabela MySQL (*SELECT id, nome, ativo FROM solenoides).*
2. **Avaliação Temporal contra Agendamentos Ativos:** Para cada solenoide (Melão, Tomate, Pimentão, Estufa), o servidor executa uma consulta SQL especial que testa se a hora atual do relógio do servidor (*CURTIME())* e a data de hoje (*CURDATE()*) estão dentro da janela de irrigação cadastrada pelo agrônomo:

```
SELECT COUNT(*) FROM agendamentos
WHERE data_agendamento = CURDATE()
  AND CURTIME() BETWEEN hora_inicio AND hora_fim
  AND solenoide = :nome
```

3. **Atualização Dinâmica do Estado do Solenoide:** Se a contagem retornar um valor maior que zero *(has_schedule > 0*), significa que aquele setor **deve estar irrigando neste exato minuto.** Se o banco de dados indicava que ele estava desligado, o servidor executa o comando SQL UPDATE solenoides *SET ativo = 1 WHERE id= :id*. Caso contrário, se o horário de irrigação já passou, o servidor atualiza para *ativo = 0.*
4. **Decisão Inteligente de Acionamento da Motobomba Principal:** O algoritmo avalia se existe qualquer solenoide ativo na fazenda ou qualquer agendamento em execução. Se houver pelo menos um setor precisando de água, a motobomba principal de 5,5 CV é automaticamente marcada para ligar (*UPDATE bomba SET ativo = 1*). Se todos os setores estiverem inativos, a motobomba é desligada (*UPDATE bomba SET ativo = 0*), protegendo a bomba contra funcionamento sem vazão e economizando energia elétrica.

Essa função garante que, mesmo que o celular do agrônomo esteja desligado ou sem bateria, a fazenda continuará sendo irrigada exatamente nos horários agendados.

6. ANÁLISE DETALHADA DAS ROTAS DE ATUADORES (bomba_routes.py)

O *arquivo bomba_routes.py* (localizado em /sistema/backend-irriga/routes/bomba_ rou tes.py) agrupa todas as rotas responsáveis pelo controle dos equipamentos físicos (moto-bomba e válvulas solenoides).

A Listagem 2 apresenta a implementação completa do módulo de rotas de atuadores.

Listing 5: Código-fonte das rotas de controle de bomba e solenoides (bomba_routes.py).

```
from flask import Blueprint, jsonify, request
from sqlalchemy import text
from db import db



bomba_routes = Blueprint('bomba_routes', __name__)



def sincronizar_status():
    solenoides = db.session.execute(text("SELECT id, nome, ativo FROM solenoides")).fetchall()



    any_solenoide_ativo = False
    for s_id, s_nome, s_ativo in solenoides:
        sql_check = """
        SELECT COUNT(*) FROM agendamentos
        WHERE data_agendamento = CURDATE()
          AND CURTIME() BETWEEN hora_inicio AND hora_fim AND
          solenoide = :nome
        """
        has_schedule = db.session.execute(text(sql_check), {"nome": s_nome}).scalar() or 0
        deve_estar_ativo = has_schedule > 0

        if deve_estar_ativo != bool(s_ativo):
            db.session.execute(
```

```
                    text("UPDATE solenoides SET ativo = :ativo WHERE id = :id"),
                    {"ativo": 1 if deve_estar_ativo else 0, "id": s_id}

                )


            if deve_estar_ativo:
                any_solenoide_ativo = True

        bomba = db.session.execute(text("SELECT id, ativo FROM bomba LIMIT 1")).fetchone()
        if bomba:
            id_bomba, bomba_ativo = bomba[0], bool(bomba[1])

            sql_agendamentos = """
            SELECT COUNT(*) FROM agendamentos
            WHERE data_agendamento = CURDATE()
              AND CURTIME() BETWEEN hora_inicio AND hora_fim
            """
            agendamentos_ativos = db.session.execute(text(sql_agendamentos)).scalar() or 0

            deve_ligar_bomba = (agendamentos_ativos > 0) or any_solenoide_ativo

            if deve_ligar_bomba != bomba_ativo:
                db.session.execute(
                    text("UPDATE bomba SET ativo = :ativo WHERE id = :id"),
                    {"ativo": 1 if deve_ligar_bomba else 0, "id": id_bomba}
                )

        db.session.commit()

@bomba_routes.route('/bomba', methods=['GET'])
def get_bomba():
    try:
        sincronizar_status()
        sql = "SELECT id, nome, ativo FROM bomba LIMIT 1"
        row = db.session.execute(text(sql)).fetchone()
        if row:
            return jsonify({"id": row[0], "nome": row[1], "ativo": bool(row[2])}), 200
        else:
            return jsonify({"error": "Bomba nao encontrada"}), 404
    except Exception as e:
        print(e)
        return jsonify({"error": "Erro interno"}), 500
```

```
@bomba_routes.route('/bomba', methods=['POST'])
def set_bomba():
    try:
        ativo = request.args.get('ativo')
        if ativo is None:
            return jsonify({"error": "Parametro 'ativo' obrigatorio"}), 400
        ativo_bool = ativo.lower() == 'true'
        sql = "UPDATE bomba SET ativo = :ativo WHERE id = 1"
        db.session.execute(text(sql), {"ativo": ativo_bool})
        db.session.commit()
        return jsonify({"message": "Bomba atualizada", "ativo": ativo_bool}), 200
    except Exception as e:
        print(e)
        return jsonify({"error": "Erro interno"}), 500

@bomba_routes.route('/solenoides', methods=['GET'])
def get_solenoides():
    try:
        sql = "SELECT id, nome, ativo FROM solenoides"
        rows = db.session.execute(text(sql)).fetchall()
        resultado = [{"id": row[0], "nome": row[1], "ativo": bool(row[2])} for row in rows]
        return jsonify(resultado), 200
    except Exception as e:
        print(e)
        return jsonify({"error": "Erro interno"}), 500

@bomba_routes.route('/solenoide', methods=['POST'])
def set_solenoide():
    try:
        id_solenoide = request.args.get('id')
        ativo = request.args.get('ativo')
        if id_solenoide is None or ativo is None:
            return jsonify({"error": "Parametros 'id' e 'ativo' sao obrigatorios"}), 400
        ativo_bool = ativo.lower() == 'true'
        sql = "UPDATE solenoides SET ativo = :ativo WHERE id = :id"
        db.session.execute(text(sql), {"ativo": ativo_bool, "id": id_solenoide})
        db.session.commit()
        return jsonify({"message": f"Solenoide {id_solenoide} atualizado", "ativo": ativo_bool}), 200
    except Exception as e:
        print(e)
        return jsonify({"error": "Erro interno"}), 500

```

```
@bomba_routes.route('/estado-geral', methods=['GET'])
def estado_geral():
    try:
        sincronizar_status()
        solenoides = db.session.execute(text("SELECT id, ativo FROM solenoides")).fetchall() bomba
        = db.session.execute(text("SELECT ativo FROM bomba WHERE id = 1")).scalar() return
        jsonify({
            "bomba": bool(bomba),
            "solenoides": [{"id": s[0], "ativo": bool(s[1])} for s in solenoides]
        }), 200
    except Exception as e:
```

6.1. O Endpoint */estado-geral*: A Rota de Consumo dos Microcontroladores

A rota mais importante para o funcionamento da Internet das Coisas na lavoura é o end-point */estado-geral* (linhas 150 a 174 da Listagem 2).

Quando o cliente magro ESP32 (Capítulo 5) dispara uma requisição *GET http://www.meusitedeirrigacao.com/estado-geral*, o servidor Flask aciona primeiro a função *sincronizar_status(),* garantindo que o banco de dados reflita o horário exato. Em seguida, o servidor lê o estado da bomba e das quatro solenoides e retorna a resposta JSON compacta:

```
{
  "bomba": true,
  "solenoides": [
    { "id": 1, "ativo": true },
    { "id": 2, "ativo": false },
    { "id": 3, "ativo": false },
    { "id": 4, "ativo": false }
  ]
}
```

Esse payload JSON consome menos de 150 bytes de banda de rede, permitindo que o ESP32 interprete os dados em milissegundos mesmo em redes Wi-Fi rurais instáveis.

7. ANÁLISE DETALHADA DAS ROTAS DE AGENDAMENTO (agendamento_routes.py)

O arquivo *agendamento_routes.py* (localizado em /sistema/backend-irriga/routes/agend amento_routes.py) gerencia todas as operações de criação, leitura, atualização e deleção (*CRUD*) das rotinas de rega cadastradas pelos operadores humanos.

A Listagem 3 apresenta o código-fonte completo das rotas de agendamento.

Listing 6: Código-fonte completo das rotas de gestão de agendamentos (agendamento_routes.py).

```
import datetime
from flask import Blueprint, jsonify, request
from sqlalchemy import text
from db import db



agendamento_routes = Blueprint('agendamento_routes', __name__)



def format_row(row):
    id, data_agendamento, hora_inicio, hora_fim, duracao_minutos, solenoide, usuario, status = row
    date_str = data_agendamento.isoformat() if hasattr(data_agendamento, 'isoformat') else str(data_agendamento)



    def format_time(t):
        if hasattr(t, 'strftime'):
            return t.strftime('%H:%M:%S')
        if isinstance(t,
            datetime.timedelta): seconds
            = t.total_seconds() hours =
            int(seconds // 3600)

            minutes = int((seconds % 3600) // 60)
            secs = int(seconds % 60)

            return f"{hours:02d}:{minutes:02d}:{secs:02d}"
        return str(t)
```

```
    }

@agendamento_routes.route('/agendamentos', methods=['GET'])
def get_agendamentos():
    try:
        sql = """
        SELECT id, data_agendamento, hora_inicio, hora_fim, duracao_minutos, solenoide, usuario, status
        FROM agendamentos ORDER BY data_agendamento DESC, hora_inicio DESC
        """
        rows = db.session.execute(text(sql)).fetchall()
        result = [format_row(row) for row in rows]
        return jsonify(result), 200
    except Exception as e:
        print(e)
        return jsonify({"error": f"Erro interno: {str(e)}"}), 500

@agendamento_routes.route('/agendamentos/<int:id>', methods=['GET'])
def get_agendamento(id):
    try:
        sql = "SELECT id, data_agendamento, hora_inicio, hora_fim, duracao_minutos, solenoide, usuario, status
            FROM agendamentos WHERE id = :id"
        row = db.session.execute(text(sql), {"id": id}).fetchone()
        if row:
            return jsonify(format_row(row)), 200
        return jsonify({"error": "Agendamento nao encontrado"}), 404
    except Exception as e:
        print(e)
        return jsonify({"error": f"Erro interno: {str(e)}"}), 500

@agendamento_routes.route('/agendamentos', methods=['POST'])
def create_agendamento():
    try:
        data = request.get_json(silent=True) or {}
        if not data:
            data = request.form.to_dict() or request.args.to_dict()

        data_agendamento = data.get('data_agendamento') or data.get('date')
        hora_inicio = data.get('hora_inicio') or data.get('startTime')
        hora_fim = data.get('hora_fim') or data.get('endTime')
        duracao_minutos = data.get('duracao_minutos') or data.get('durationMinutes')
        solenoide = data.get('solenoide') or data.get('solenoid')
        usuario = data.get('usuario') or data.get('scheduledBy')
```

```
        status = data.get('status') or 'AGENDADO'

        if not all([data_agendamento, hora_inicio, hora_fim, duracao_minutos, solenoide, usuario]):
            return jsonify({"error": "Campos obrigatorios faltando"}), 400

        sql = """
        INSERT INTO agendamentos (data_agendamento, hora_inicio, hora_fim, duracao_minutos, solenoide,
            usuario, status)
        VALUES (:data_agendamento, :hora_inicio, :hora_fim, :duracao_minutos, :solenoide, :usuario, :status)
        """
        result = db.session.execute(text(sql), {
            "data_agendamento": data_agendamento,
            "hora_inicio": hora_inicio,
            "hora_fim": hora_fim,
            "duracao_minutos": int(duracao_minutos),
            "solenoide": solenoide,
            "usuario": usuario,
            "status": status
        })
        db.session.commit()

        new_id = result.lastrowid
        if not new_id:
            new_id_row = db.session.execute(text("SELECT LAST_INSERT_ID()")).fetchone()
            new_id = new_id_row[0] if new_id_row else None

        return jsonify({
            "message": "Agendamento criado com sucesso",
            "id": new_id,
            "data_agendamento": data_agendamento,
            "hora_inicio": hora_inicio,
            "hora_fim": hora_fim,
            "duracao_minutos": int(duracao_minutos),
            "solenoide": solenoide,
            "usuario": usuario,
            "status": status
        }), 201
    except Exception as e:
        db.session.rollback()
        print(e)
        return jsonify({"error": f"Erro interno: {str(e)}"}), 500
```

```
@agendamento_routes.route('/agendamentos/<int:id>', methods=['PUT', 'POST'])
def update_agendamento(id):
    try:
        check_sql = "SELECT id FROM agendamentos WHERE id = :id"
        exists = db.session.execute(text(check_sql), {"id": id}).fetchone()
        if not exists:
            return jsonify({"error": "Agendamento nao encontrado"}), 404

        data = request.get_json(silent=True) or {}
        if not data:
            data = request.form.to_dict() or request.args.to_dict()

        data_agendamento = data.get('data_agendamento') or data.get('date')
        hora_inicio = data.get('hora_inicio') or data.get('startTime')
        hora_fim = data.get('hora_fim') or data.get('endTime')
        duracao_minutos = data.get('duracao_minutos') or data.get('durationMinutes')
        solenoide = data.get('solenoide') or data.get('solenoid')
        usuario = data.get('usuario') or data.get('scheduledBy')
        status = data.get('status')

        updates = []
        params = {"id": id}

        if data_agendamento is not None:
            updates.append("data_agendamento = :data_agendamento")
            params["data_agendamento"] = data_agendamento
        if hora_inicio is not None:
            updates.append("hora_inicio = :hora_inicio")
            params["hora_inicio"] = hora_inicio
        if hora_fim is not None:
            updates.append("hora_fim = :hora_fim")
            params["hora_fim"] = hora_fim
        if duracao_minutos is not None:
            updates.append("duracao_minutos = :duracao_minutos")
            params["duracao_minutos"] = int(duracao_minutos)
        if solenoide is not None:
            updates.append("solenoide = :solenoide")
            params["solenoide"] = solenoide
        if usuario is not None:
            updates.append("usuario = :usuario")
            params["usuario"] = usuario
        if status is not None:
```

```
            updates.append("status =
            :status") params["status"] =
            status

        if not updates:
            return jsonify({"message": "Nenhuma alteracao enviada"}), 200

        sql = f"UPDATE agendamentos SET {', '.join(updates)} WHERE id = :id"
        db.session.execute(text(sql), params)
        db.session.commit()
        return jsonify({"message": "Agendamento atualizado com sucesso"}), 200
    except Exception as e:
        db.session.rollback()
        print(e)
        return jsonify({"error": f"Erro interno: {str(e)}"}), 500

@agendamento_routes.route('/agendamentos/<int:id>', methods=['DELETE'])
def delete_agendamento(id):
    try:
        check_sql = "SELECT id FROM agendamentos WHERE id = :id" exists
        = db.session.execute(text(check_sql), {"id": id}).fetchone() if not
        exists:
            return jsonify({"error": "Agendamento nao encontrado"}), 404
```

7.1. A Função Auxiliar de Formatação de Dados *format_row()*

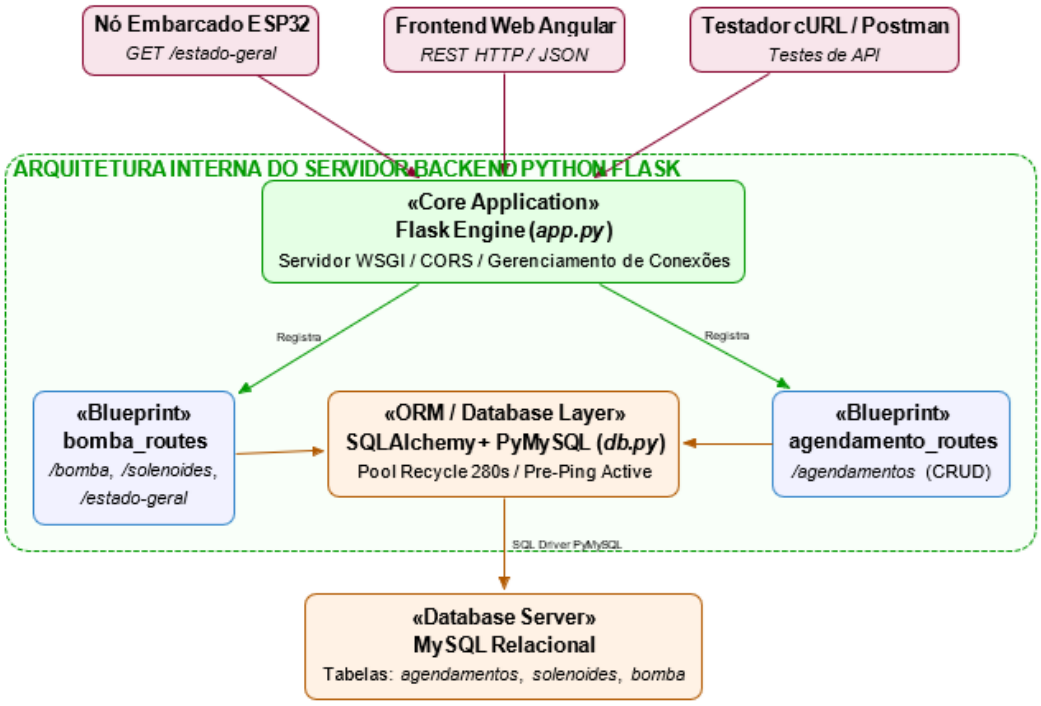
Ao consultar linhas no MySQL relacional, o driver PyMySQL devolve tuplas contendo objetos nativos de data (*datetime.date*) e tempo (*datetime.timedelta*). Como o formato de texto JSON não possui suporte nativo a objetos de data do Python, a função *format_row()* (linhas 8 a 34 da Listagem 3) realiza a conversão padronizada:

- Converte o objeto de data para a string padronizada ISO 8601 (*AAAA-MM-DD*);
- Calcula as horas, minutos e segundos a partir de objetos *timedelta*, devolvendo a string formatada *HH:MM:SS;*
- Garante que o documento JSON retornado pelo backend possua uma estrutura homogênea e amigável para a aplicação web em Angular.

8. MODELAGEM ARQUITETURAL DO BACKEND VIA DIAGRAMA DE COMPONENTES UML

Para formalizar a estrutura interna do servidor backend sob a ótica da engenharia de software, a Figura 1 apresenta o **Diagrama de Componentes UML** da aplicação Python Flask.

Figura 22: Diagrama de Componentes UML do servidor backend Flask em irrigação IoT.



Como ilustrado no diagrama da Figura 1, o servidor Flask atua como o ponto central de roteamento: ele recebe as requisições enviadas pelos clientes externos, encaminha o fluxo para o Blueprint especialista correspondente e executa as rotinas SQL seguras via SQLAlchemy na base de dados MySQL.

9. FERRAMENTAS DE TESTE DE APIS: O QUE SÃO cURL E POSTMAN?

Durante a fase de implantação ou manutenção de um sistema de irrigação automatizado, é fundamental verificar se o servidor backend está funcionando corretamente antes de conectar os painéis de campo ESP32 ou publicar a aplicação web. Se o sistema falhar, a equipe precisa saber se o problema está na fiação elétrica do campo, na interface web ou na API backend.

Para testar e validar o funcionamento das rotas HTTP da API de forma isolada, os engenheiros de software utilizam duas ferramentas indispensáveis: o **cURL** e o **Postman**.

9.1. O que é o cURL (*Client URL*)?

O cURL é uma ferramenta utilitária de linha de comando integrada nativamente a sistemas operacionais modernos (Linux, macOS e Windows Terminal).

Para o agrônomo, o cURL é o equivalente a um multímetro digital de rede: uma ferramenta simples e extremamente rápida que permite ao técnico enviar uma requisição HTTP crua diretamente do terminal de comandos para o servidor *www.meusitedeirrigacao.com* e ver na tela exatamente o texto JSON devolvido pelo Flask, juntamente com o código de status (ex: 200 OK ou 400 Bad Request).

9.2. O que é o Postman?

O **Postman** é um ambiente gráfico de desenvolvimento e testes de APIs.

Enquanto o cURL é utilizado via texto na linha de comando, o Postman oferece uma interface visual completa com botões, caixas de diálogo, abas para preenchimento de parâmetros, salvamento de coleções de testes e visualização formatada e colorida dos payloads JSON retornados pelo servidor. O Postman permite ao engenheiro simular visualmente a atuação de uma solenoide, cadastrar agendamentos de teste e validar respostas em segundos.

10. EXEMPLOS PRÁTICOS DE TESTES DE ROTAS COM cURL E POSTMAN

A seguir, apresentam-se exemplos práticos e reais de comandos cURL e requisições no Postman para testar cada uma das rotas do backend publicado em *www.meusitedeirrigacao.com*.

10.1. Teste 1: Consulta do Estado Geral (Consumo do ESP32)

- **Objetivo:** Verificar se a rota */estado-geral* está respondendo corretamente e atualizando a bomba e solenoides com base no horário atual.
- **Comando cURL no Terminal:**

```
curl -X GET "http://www.meusitedeirrigacao.com/estado-geral"
```

- **Configuração no Postman:**
  - Método: *GET*
  - URL: http://www.meusitedeirrigacao.com/estado-geral
  - Clique no botão **Send**.

- **Resposta Esperada do Servidor (HTTP 200 OK):**

```
{
  "bomba": true,
  "solenoides": [
    { "id": 1, "ativo": true },
    { "id": 2, "ativo": false },
    { "id": 3, "ativo": false },
    { "id": 4, "ativo": false }
  ]
}
```

10.2. Teste 2: Cadastro de Novo Agendamento de Irrigação

- **Objetivo:** Inserir uma nova rotina de rega para o setor do Tomate com duração de 20 minutos.
- **Comando cURL no Terminal:**

```
curl -X POST "http://www.meusitedeirrigacao.com/agendamentos" \
     -H "Content-Type: application/json" \
     -d '{
           "data_agendamento": "2026-09-30",
           "hora_inicio": "08:00:00",
           "hora_fim": "08:20:00",
           "duracao_minutos": 20,
           "solenoide": "Tomate",
           "usuario": "Thiago (Agronomo)"
         }'
```

- **Configuração no Postman:**
  - Método: *POST*
  - URL: http://www.meusitedeirrigacao.com/agendamentos
  - Aba **Headers**: Adicionar *Content-Type: application/json*
  - Aba **Body**: Selecionar a opção raw (formato JSON) e colar o objeto com os dados da rega.
  - Clique no botão **Send**.
- **Resposta Esperada do Servidor (HTTP 201 Created):**

```
{
  "id": 15,
  "message": "Agendamento criado com sucesso",
  "data_agendamento": "2026−09−30",
  "hora_inicio": "08:00:00",
  "hora_fim": "08:20:00",
  "duracao_minutos": 20,
  "solenoide": "Tomate",
  "usuario": "Thiago (Agronomo)",
  "status": "AGENDADO"
}
```

10.3. Teste 3: Acionamento Manual de Válvula Solenoide

- **Objetivo:** Forçar a abertura imediata da solenoide 1 (Setor Melão).
- **Comando cURL no Terminal:**

```
curl −X POST "http://www.meusitedeirrigacao.com/solenoide?id=1&ativo=true"
```

- **Configuração no Postman:**
  - Método: *POST*
  - URL: http://www.meusitedeirrigacao.com/solenoide?id=1&ativo=true
  - Clique no botão **Send**.

- **Resposta Esperada do Servidor (HTTP 200 OK):**

```
{
  "ativo": true,
  "message": "Solenoide 1 atualizado"
}
```

10.4. Teste 4: Atualização de Agendamento Existente

- **Objetivo:** Alterar a duração e o horário de término do agendamento ID 15.
- **Comando cURL no Terminal:**

```
curl −X PUT "http://www.meusitedeirrigacao.com/agendamentos/15" \
     −H "Content−Type: application/json" \
     −d '{
           "hora_fim": "08:30:00",
           "duracao_minutos": 30
         }'
```

- **Configuração no Postman:**
  - Método: *PUT*
  - URL: http://www.meusitedeirrigacao.com/agendamentos/15
  - Aba **Body**: Formato JSON contendo apenas os campos atualizados.
  - Clique no botão **Send**.
- **Resposta Esperada do Servidor (HTTP 200 OK):**

```
{
  "message": "Agendamento atualizado com sucesso"
}
```

10.5. Teste 5: Deleção de Agendamento Cancelado

- **Objetivo:** Excluir o agendamento ID 15 do banco de dados MySQL.
- **Comando cURL no Terminal:**

```
curl -X DELETE "http://www.meusitedeirrigacao.com/agendamentos/15"
```

- **Configuração no Postman:**
  - Método: *DELETE*
  - URL: http://www.meusitedeirrigacao.com/agendamentos/15
  - Clique no botão **Send**.
- **Resposta Esperada do Servidor (HTTP 200 OK):**

```
{
  "message": "Agendamento deletado com sucesso"
}
```

11. CONSIDERAÇÕES DE SEGURANÇA E DEPLOYMENT EM AMBIENTES DE PRODUÇÃO RURAL

Para garantir que o backend Python Flask opere com máxima estabilidade e segurança na infraestrutura de servidores da fazenda digital, devem-se observar as seguintes práticas de engenharia:

- **Servidor WSGI de Produção:** O servidor embutido do *Flask (app.run())* é voltado apenas para desenvolvimento e testes. Em produção no servidor corporativo *www.meusitedeirrigacao.com*, a aplicação deve ser executada através de um servidor WSGI profissional (como o Gunicorn ou uWSGI), gerenciado por um servidor web Nginx ou pelo utilitário *passenger_wsgi.py.*
- **Proteção CORS Restritiva:** Em ambiente de produção, a diretiva *CORS(app)* deve ser configurada para permitir requisições apenas vindo da origem autorizada do painel web da fazenda, impedindo que sites não autorizados na internet façam

requisições maliciosas para acionar solenoides no campo.

- **Variáveis de Ambiente para Credenciais:** As senhas do banco de dados MySQL e a chave secreta da aplicação *(app.secret_key*) nunca devem ser gravadas diretamente em código-fonte aberto, devendo ser injetadas dinamicamente através de variáveis de ambiente do sistema operacional *(os.environ).*

12. SÍNTESE DO CAPÍTULO E CONCLUSÃO GERAL DA OBRA

Este capítulo apresentou a arquitetura completa do servidor **Backend Python Flask**, que atua como o cérebro central de processamento e persistência do sistema hipotético de irrigação automatizada em Internet das Coisas.

Ficou demonstrada a importância da modularização de rotas via Blueprints, a gestão resiliente do banco de dados relacional MySQL através do SQLAlchemy com *pool recycling* e *pre-ping*, e o funcionamento do algoritmo inteligente da função *sincronizar_status()*, que avalia os agendamentos temporais e atualiza automaticamente os relés e a motobomba principal. Além disso, a arquitetura foi formalizada via Diagrama de Componentes UML e os testes práticos de rotas foram detalhados utilizando as ferramentas cURL e Postman no domínio corporativo *www.meusitedeirrigacao.com.*

Com o encerramento deste sétimo capítulo, esta obra consagra-se como um tratado abrangente, rigoroso e prático sobre a engenharia de sistemas de irrigação automatizados com IoT:

- **Capítulo 1:** Introdução às interfaces web e ecossistema de irrigação IoT;
- **Capítulo 2:** Especificações dos equipamentos, sensores capacitivos, válvulas solenoides e motobombas;
- **Capítulo 3:** Circuitos eletreletrônicos de amostragem, optoacopladores e chaves contatoras de potência;
- **Capítulo 4:** Fundamentos do protocolo HTTP REST e arquitetura cliente-servidor;
- **Capítulo 5:** Firmware C++ em microcontrolador ESP32, modelo de Cliente Magro (*Thin Client*) de Tanenbaum e salvaguardas falha-segura;
- **Capítulo 6:** Arquitetura Frontend SPA Angular, Programação Orientada a Objetos, Diagrama de Componentes UML, desacoplamento de backend e ecossistemas multicliente;
- **Capítulo 7:** Servidor Backend Python Flask, arquitetura de rotas RESTful, sincronização automática temporal, persistência MySQL relacional e testes de integração com cURL e Postman.

REFERÊNCIAS

MARTIN, Robert C. **Clean Architecture: A Craftsman's Guide to Software Structure and Design**. Boston: Prentice Hall, 2017.

SOMMERVILLE, Ian. **Software Engineering**. 10. ed. Boston: Pearson, 2016.