

# Documentação Técnica – Estacionamento

Documentação do projeto de Controle de Estacionamento Garen

# Sumário

1. Visão geral do sistema.....	4
1.1 Objetivo.....	4
2. Arquitetura e especificação do sistema.....	4
2.1 Visão geral da arquitetura.....	4
2.2 Componentes Principais.....	5
2.2.1 Camada de Hardware.....	5
2.2.2 Camada de Software.....	5
2.2.3 Camada de Dados.....	5
2.3 Requisitos de Operação.....	6
2.4 Requisitos de Software.....	6
3 Atualização e configuração.....	6
3.1 Arquivos para Atualização.....	6
3.1.1 updater.py.....	7
3.1.2 updateVerifier.py.....	7
3.2 Arquivo de configuração de regras.....	8
3.2.1 Parâmetros de Rede.....	9
3.2.2 Credenciais e Integração.....	9
3.3 Conexão com a placa via SSH.....	9
4. Estruturação banco de dados.....	10
5. Software Controle de Estacionamento.....	15
5.1 Dashboard.....	15
5.2 Dispositivos.....	16
5.3 Andares.....	17
5.4 Layout dos andares e sincronização.....	18
5.5 Display.....	20
5.6 Usuários.....	20
6. Código fonte e Comunicação.....	21
6.1. Estrutura do projeto.....	21
6.2 Comunicação.....	22

6.2.1 Software de estacionamento.....	22
6.2.1 API REST.....	22
6.2.2 Comunicação Serial.....	22
6.2.2.1 descrição enviaComandos.py.....	23
6.2.2.2 descrição recebeEventos.py.....	24
7. Endpoints.....	25
7.1 Autenticação.....	25
7.2 Configuração.....	26
7.3 Dashboard.....	27
7.4 Display.....	33
7.4 Andares.....	35
7.5 Vagas.....	36

# 1. Visão geral do sistema

O sistema de gerenciamento de estacionamento é uma solução integrada para monitoramento e controle de vagas em tempo real, oferecendo análises detalhadas sobre o uso do estacionamento e ferramentas de configuração visual.

## 1.1 Objetivo

Proporcionar controle total sobre a ocupação de vagas, oferecendo dados em tempo real e históricos para otimização da operação e melhor experiência do usuário final.

# 2. Arquitetura e especificação do sistema

## 2.1 Visão geral da arquitetura.

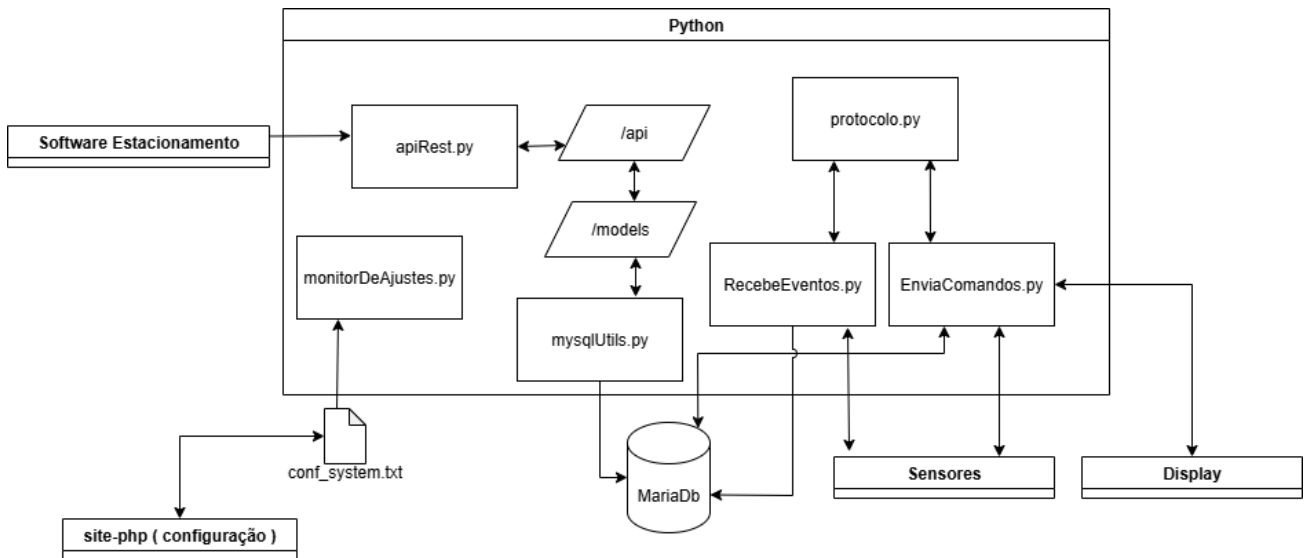


Figura 1: Diagrama de comunicação

Fluxo principal de dados:

1. Placa controladora requisita status aos sensores conectados.
2. Sensores respondem status sobre presença de veículos nas vagas
3. Sinais são transmitidos via RS-485 para a placa controladora
4. Placa controladora interpreta os sinais e atualiza o status das vagas
5. Sistema processa os dados e atualiza displays e dashboard em tempo real
6. Dados históricos são armazenados para análises posteriores

## 2.2 Componentes Principais

O sistema é composto por três camadas principais:

### 2.2.1 Camada de Hardware

- **Sensores:** Dispositivos instalados em cada vaga para detecção de ocupação
- **Placa Controladora:** Unidade central que recebe e interpreta os sinais dos sensores
- **Displays:** Painéis de exibição para informações em tempo real

### 2.2.2 Camada de Software

- **Software Estacionamento:** Interface para monitoramento e configuração do sistema de estacionamento.
- **site-php** interface de configuração de rede.

### 2.2.3 Camada de Dados

- **MariaDb** Armazenamento de eventos de sensores, configurações e usuarios
- **conf\_system.txt** arquivo de configuração de rede

## 2.3 Requisitos de Operação

- **Alimentação:** 12V DC.
- **Conectividade de rede:** Cabo de rede padrão **Ethernet (RJ45)** para configuração inicial.
- **Configuração inicial:** A placa requer a Configuração dos parâmetros de rede e no software para seu funcionamento

## 2.4 Requisitos de Software

- **Python:** Versão 3.9.9
- **MariaDB:** Versão 10.3.34
- **Dependências Python:**

```
# Dependências principais do projeto
Flask==2.3.3
Flask-CORS==4.0.0
Flask-RESTX==1.2.0

# Banco de dados
mysqlclient==2.2.0

# Compressão/Zip
pyzipper==0.3.6

# HTTP requests
requests==2.31.0

# Utilitários
Werkzeug==2.3.7
```

## 3 Atualização e configuração

### 3.1 Arquivos para Atualização

A atualização da placa controladora é feita utilizando 2 arquivos, que serão descritos logo abaixo. Um dos arquivos é o **gerador**, responsável por criar o pacote **update.bin**, reunindo os diretórios e arquivos necessários, aplicando criptografia e compressão para garantir segurança no transporte. O segundo arquivo é o **executador**, embarcado na própria placa, que identifica o pacote **update.bin**, realiza a extração dos dados, substitui os diretórios antigos e aplica as atualizações no sistema e no banco de dados de forma automatizada.

### 3.1.1 updater.py

Script responsável para criar o pacote de atualização do sistema, reunindo arquivos e pastas essenciais, como banco de dados, scripts e diretórios do software. O conteúdo é compactado e criptografado com senha, garantindo segurança na distribuição. O arquivo gerado (**update.bin**) é utilizado para aplicar atualizações de forma automatizada e protegida.

```
def zip_dir(path, _zip):
    for root, dirs, files in os.walk(path):
        for file in files:
            _zip.write(os.path.join(root, file),
                       os.path.relpath(os.path.join(root, file), os.path.join(path, '..')))

def zip_web_dir(path, _zip):
    target_folders = ['assets', 'css', 'views']
    estacionamento_path = os.path.join(path, 'estacionamento')
    if not os.path.exists(estacionamento_path):
        print(f"Diretório {estacionamento_path} não encontrado!")
        return
    for folder in target_folders:
        folder_path = os.path.join(estacionamento_path, folder)
        if os.path.exists(folder_path):
            for root, dirs, files in os.walk(folder_path):
                if folder == 'assets' and 'libs' in dirs:
                    dirs.remove('libs')

                for file in files:
                    file_path = os.path.join(root, file)
                    arcname = os.path.relpath(file_path, path)
                    _zip.write(file_path, arcname)
            print(f"Pasta {folder} adicionada ao zip")
        else:
            print(f"Pasta {folder_path} não encontrada!")

with pyzipper.AESZipFile('update.bin', 'w', compression=pyzipper.ZIP_LZMA, encryption=pyzipper.WZ_AES) as _zip:
    _zip.setpassword(password)
    for _dir in dir_list:
        zip_dir(_dir, _zip)

    zip_web_dir(r'E:\xampp\htdocs', _zip)
    _zip.write('updateVerifier.py')
    _zip.close()
```

Figura 2: Script que gera o arquivo de atualização.

### 3.1.2 updateVerifier.py

Script responsável por aplicar a atualização do sistema diretamente na placa. Ele verifica a existência dos arquivos **update.bin** e **update\_senior.bin**, realiza a extração segura com senha, substitui os diretórios antigos (backup, regras, site-php, senior) e atualiza o banco de dados com os comandos SQL. Também aplica permissões corretas e reinicia o Nginx caso necessário.

```

def copy_tree_safe(src, dst):
    src_path = Path(src)
    dst_path = Path(dst)
    dst_path.mkdir(parents=True, exist_ok=True)

    for item in src_path.rglob('*'):
        if item.is_file():
            relative_path = item.relative_to(src_path)
            dest_file = dst_path / relative_path
            dest_file.parent.mkdir(parents=True, exist_ok=True)
            shutil.copy2(item, dest_file)

with pyzipper.AESZipFile('/var/www/localhost/html/update/update.bin',
                        compression=pyzipper.ZIP_LZMA,
                        encryption=pyzipper.WZ_AES) as zf:
    zf.setpassword(password)
    zf.extractall('/home/update')
    permission = "chmod", "-R", "755", "/home/update"
    subprocess.run(permission)

if Path('/home/update').exists() \
    and Path('/home/update/backup').exists():
    if Path('/home/backup').exists():
        shutil.rmtree('/home/backup')
    shutil.move("/home/update/backup/", "/home/")
    if Path('/home/update/site-php').exists():
        copy_tree_safe('/home/update/site-php', '/home/site-php')
    if Path('/home/update/estacionamento').exists():
        copy_tree_safe('/home/update/estacionamento', '/home/site-php/estacionamento')

    shutil.copy("/home/update/updateVerifier.py", "/home/")
    shutil.rmtree('/home/update')

```

*Figura 3: Processo de extração e substituição dos diretórios a partir do arquivo update.bin*

Este trecho do script verifica se o arquivo **update.bin** está presente no diretório de atualização da placa. Se encontrado, ele é extraído com segurança usando criptografia AES e os arquivos são movidos para **/home/update**. Em seguida, os diretórios antigos são removidos e substituídos pelos novos dados extraídos. Também é copiado o arquivo **updateVerifier.py**, que será utilizado nas etapas seguintes da atualização.

## 3.2 Arquivo de configuração de regras

O Arquivo **configuracao.txt** se encontra em **/home/configs**. Este arquivo JSON define os parâmetros de rede e controle de acesso da placa. Ele é essencial para o funcionamento básico do sistema, garantindo conectividade, segurança e comportamento operacional.



### 3.2.1 Parâmetros de Rede

- **TipoIP:** Define se o IP é fixo ou dinâmico (DHCP).
- **IP, MascaraDeRede, DNS:** Configurações padrão de rede local.

### 3.2.2 Credenciais e Integração

- **weblogin, webpass:** Usuário e senha para acesso à interface web da placa.

## 3.3 Conexão com a placa via SSH

Para acessar a placa remotamente (putty ou WinSCP), deve-se utilizar a chave privada horus\_rsa.ppk

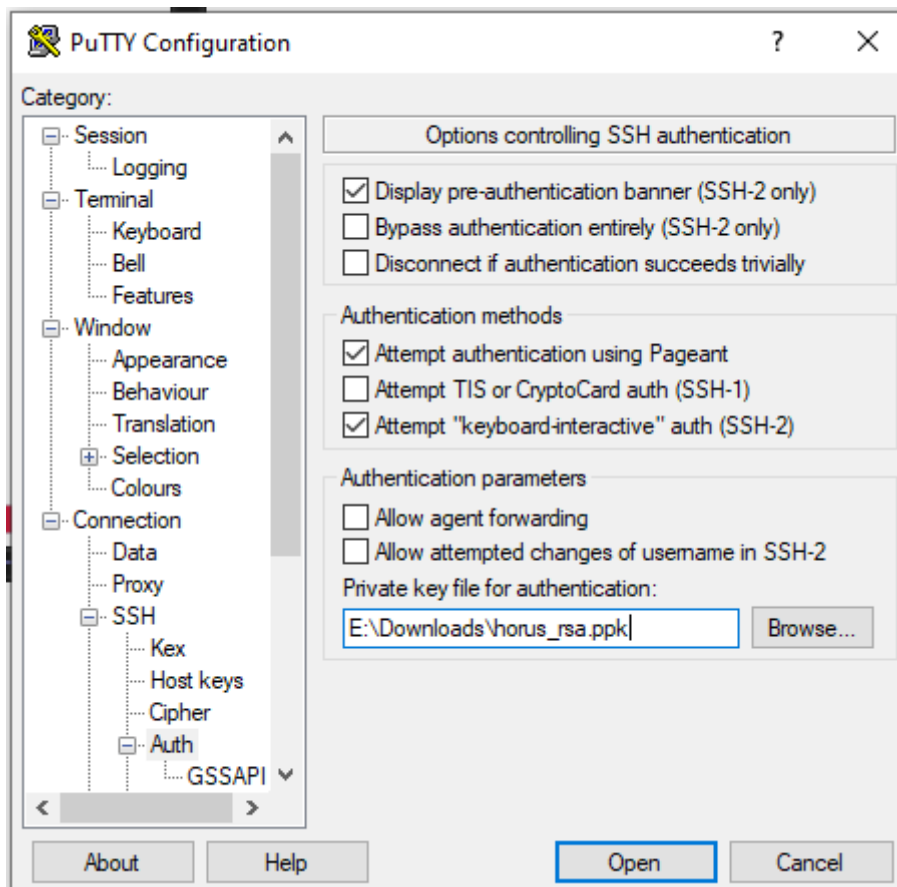


Figura 4: Tela de configuração private key PuTTY

No campo **Host Name (or IP address)**, insira o ip da placa (caso conectada por usb, 192.168.0.100) , na inicialização quando solicitado o usuário, informe **root**, não será solicitada senha, pois a autenticação é feita via chave.

## 4. Estruturação banco de dados

Aqui está descrita os tipos e tabelas existentes na controladora de acesso assim como uma breve descrição.

config			
Nome	Tipo	Key	Default
id	Int(10)	Primary	auto_increment
default_sensor_heigh	Int(11)		500
default_sensor_widht	Int(11)		500
sync_device_id	Int(11)		Null
sync_floor_id	Int(11)		Null
sync_x	Int(11)		Null
sync_y	Int(11)		Null

Utilizada para controlar as configurações padrão de sistema.

- **default\_sensor\_height:** representa a altura padrão que os sensores se encontram do chão.
- **default\_sensor\_width:** representa a largura da área que o sensor detectará.
- **sync\_device\_id:** representa o id atual do dispositivo que está sendo sincronizado.
- **sync\_floor\_id:** representa o id do andar do dispositivo que está sendo sincronizado.
- **sync\_x** e **sync\_y:** correspondem as posições horizontal e vertical do dispositivo no estacionamento

device_events			
Nome	Tipo	Key	Default
event_id	Int(11)	Primary	auto_increment
device_id	Int(11)	Multiples	Null
event_type	Varchar(11)		Null
status	Varchar(11)		Null
received_at	Timestamp	current_timestamp()	Null
additional_data	longtext		Null

A tabela armazena os eventos gerados pelos dispositivos, sendo utilizada como base para alimentar as informações exibidas no dashboard.

- **device\_id:** é o campo **id** encontrado na tabela **devices**
- **event\_type:** tipo de evento salvo (atualmente só tem status).
- **status:** Estado atual do dispositivo, podendo estar como:
  - *free*: quando o dispositivo não detecta nenhum automóvel
  - *occupied*: quando o dispositivo detecta um automóvel
  - *offline*: caso o dispositivo não tenha respondido as requisições de status
- **received\_at:** timestamp que guarda o horario que o evento foi salvo no banco.
- **additional\_data:** campo reservado para observações.

devices			
Nome	Tipo	Key	Default
id	Int(10)	Primary	auto_increment
Name	Varchar(128)		Null
Status	Varchar(32)		Null
floor_id	Int(11)	Multiples	Null
X	Int(11)		Null
Y	Int(11)		Null
sensor_height	Int(11)		Null

Representa o cadastro e status dos dispositivos utilizados no sistema.

- **Name:** nome do dispositivo (vaga) cadastrado.
- **status:** status atual do dispositivo, podendo ser:
  - *free*: quando o dispositivo não detecta nenhum automóvel
  - *occupied*: quando o dispositivo detecta um automóvel
  - *offline*: caso o dispositivo não tenha respondido as requisições de status
  - *road*: quando a vaga é mapeada como "rua" no mapa do estacionamento.
  - *block*: quando a vaga é mapeada como "bloqueado" no mapa do estacionamento.
  - *unknown*: quando a vaga foi criada mas não teve nenhuma configuração adicional ou não foi vinculada a um dispositivo.
- **floor\_id:** é o campo *id* encontrado na tabela *floors*.
- **x:** Posição vertical que o dispositivo se encontra no estacionamento.
- **y:** Posição horizontal que o dispositivo se encontra no estacionamento.
- **sensor\_height:** Altura que o sensor se encontra do chão, caso seja diferente do padrão

display_config			
Nome	Tipo	Key	Default
id	Int(11)	Primary	auto_increment
display_number	Int(11)	Multiples	
floors	Longtext		Null

Representa quais displays exibem quais andares.

- **display\_number:** é como se fosse o segundo id do display, representa o número do que precisa ser configurado no display físico para exibir os andares cadastrados.
- **floors:** representa uma lista separada por “,” dos ids dos andares que o display exibirá a quantidade de vagas.

Floors			
Nome	Tipo	Key	Default
id	Int(11)	Primary	auto_increment
floor	Varchar(80)		Null
width	Int(11)		Null
height	Int(11)		Null
x_in	Int(11)		Null
y_in	Int(11)		Null

Representa a configuração dos andares salvos no sistema.

- **floor:** é nome que será exibido do andar.
- **width:** é a largura do andar do estacionamento.
- **height:** é a altura do andar do estacionamento.
- **x\_in:** é a posição vertical da entrada do estacionamento.
- **y\_in:** é a posição horizontal da entrada do estacionamento.

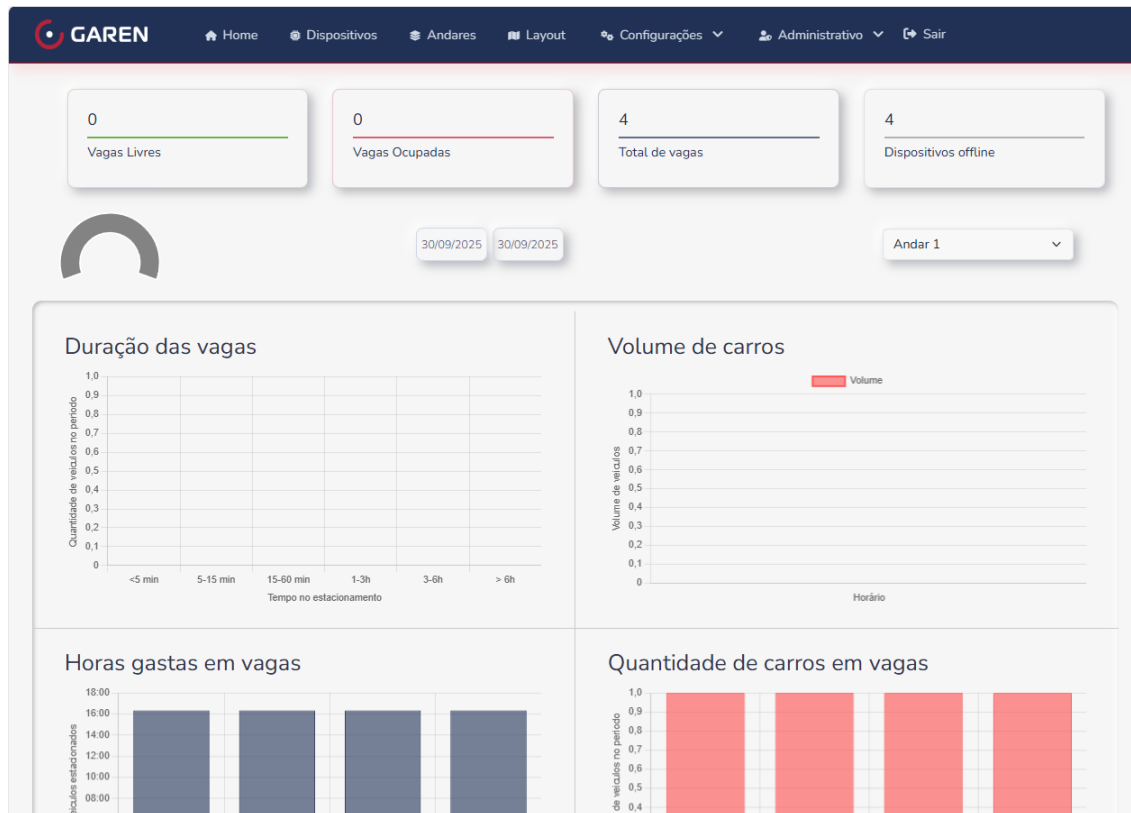
users			
Nome	Tipo	Key	Default
Id	Int(11)	Primary	auto_increment
username	Varchar(80)	Unique	Null
authority	Varchar(80)		Null
password_hash	Varchar(256)		Null
active	Int(1)		1

Tabela representando o acesso dos usuarios no sistema.

- **username:** é o campo de nome de usuario usado para realizar login no sistema.
- **authority:** é o campo de autoridade do usuario, podendo ser user, ou admin.
- **password\_hash:** é a senha criptografada definida para o usuario logar ao sistema.
- **active:** booleano responsável pela validação se o usuario está habilitado ou não a acessar o sistema.

## 5. Software Controle de Estacionamento.

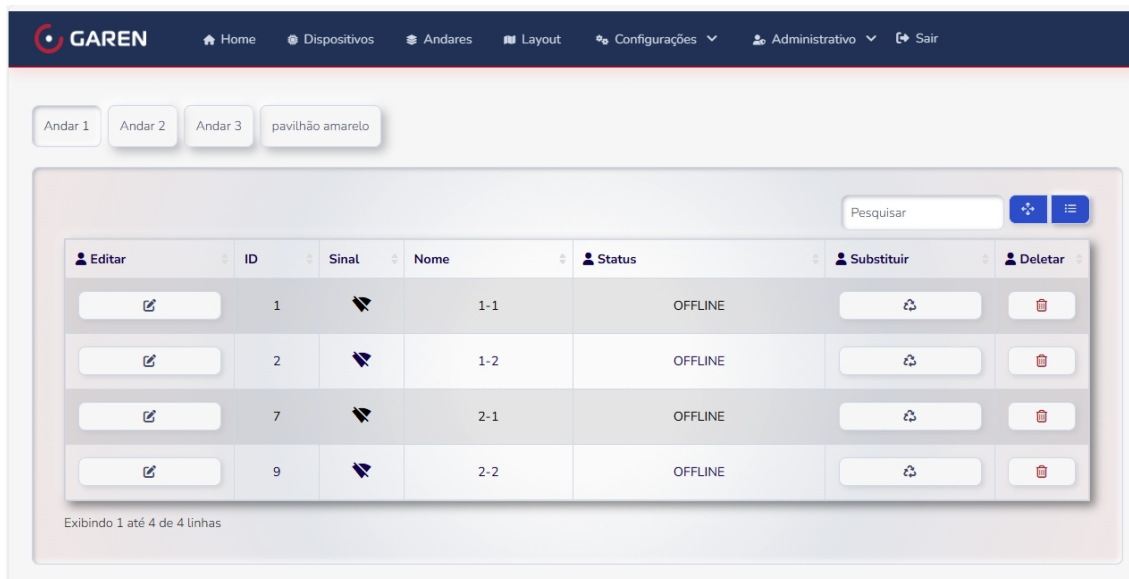
### 5.1 Dashboard



**Figura 5: Tela inicial - Dashboard**

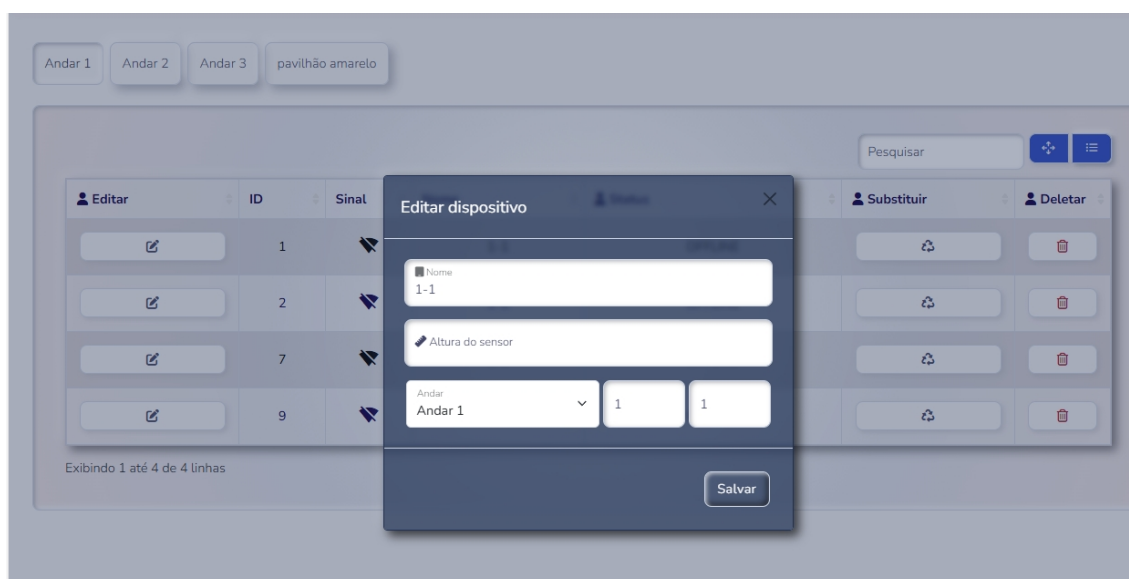
Esta tela foi desenvolvida para monitorar e gerenciar em tempo real o uso de vagas em um estacionamento inteligente. Ela oferece uma visão analítica e detalhada sobre o comportamento dos veículos.

## 5.2 Dispositivos



**Figura 6: Tela de Dispositivos**

Esta interface é responsável por exibir os dispositivos cadastrados em cada andar, é possível alternar entre os andares e pesquisar por dispositivos específicos, editar informações, ressincronizar aquele id com outro dispositivo físico, e também deletar aquele id.

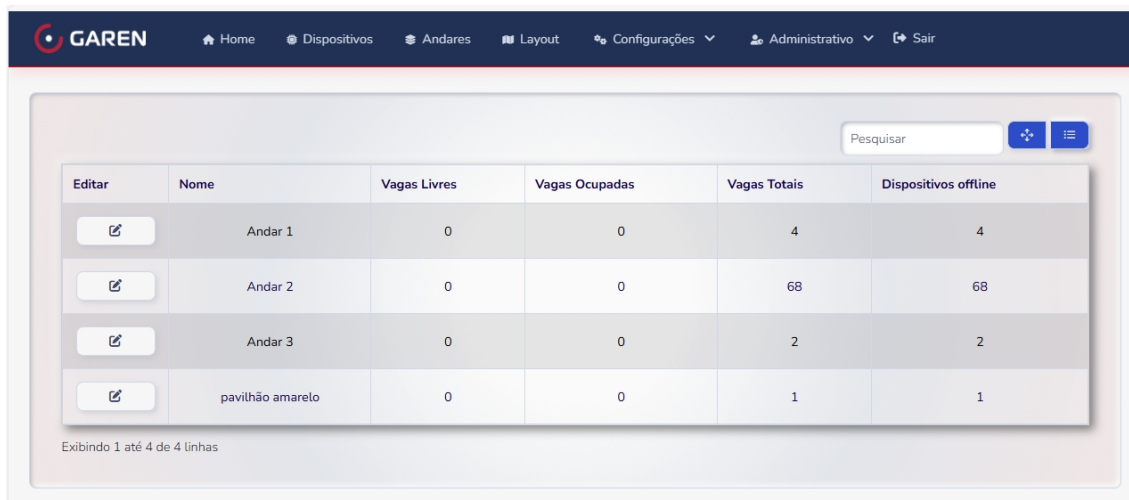


**Figura 7: Edição de dispositivo**

Editando o dispositivo é possível alterar os campos de nome, altura do sensor, e qual andar e localização que aquele sensor se encontra.



## 5.3 Andares

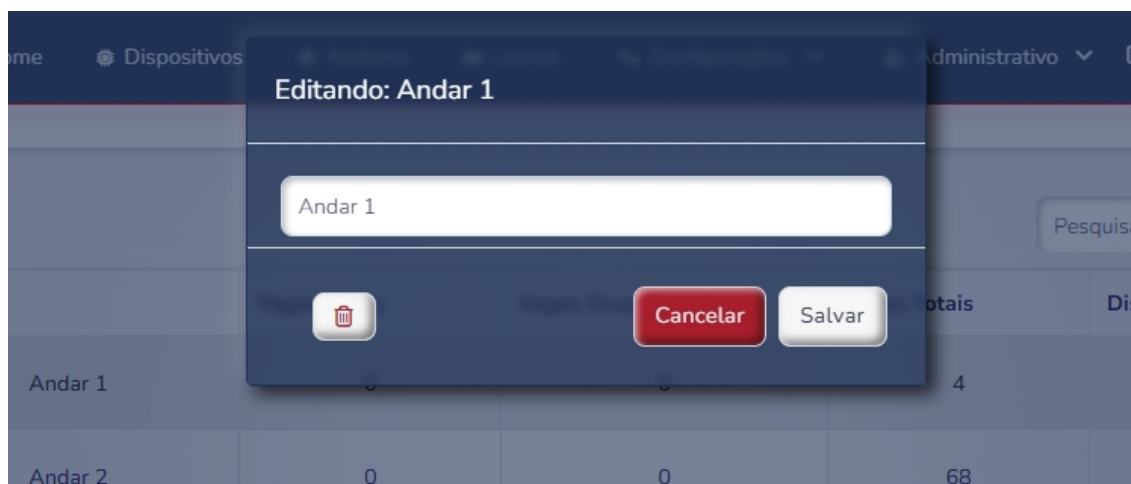


Editar	Nome	Vagas Livres	Vagas Ocupadas	Vagas Totais	Dispositivos offline
	Andar 1	0	0	4	4
	Andar 2	0	0	68	68
	Andar 3	0	0	2	2
	pavilhão amarelo	0	0	1	1

Exibindo 1 até 4 de 4 linhas

*Figura 8: Tela de Andares*

A tela de Andares permite acompanhar o status agregado de cada andar do estacionamento. Ela apresenta uma tabela interativa com indicadores por andar, e também é possível fazer edição do nome do andar.



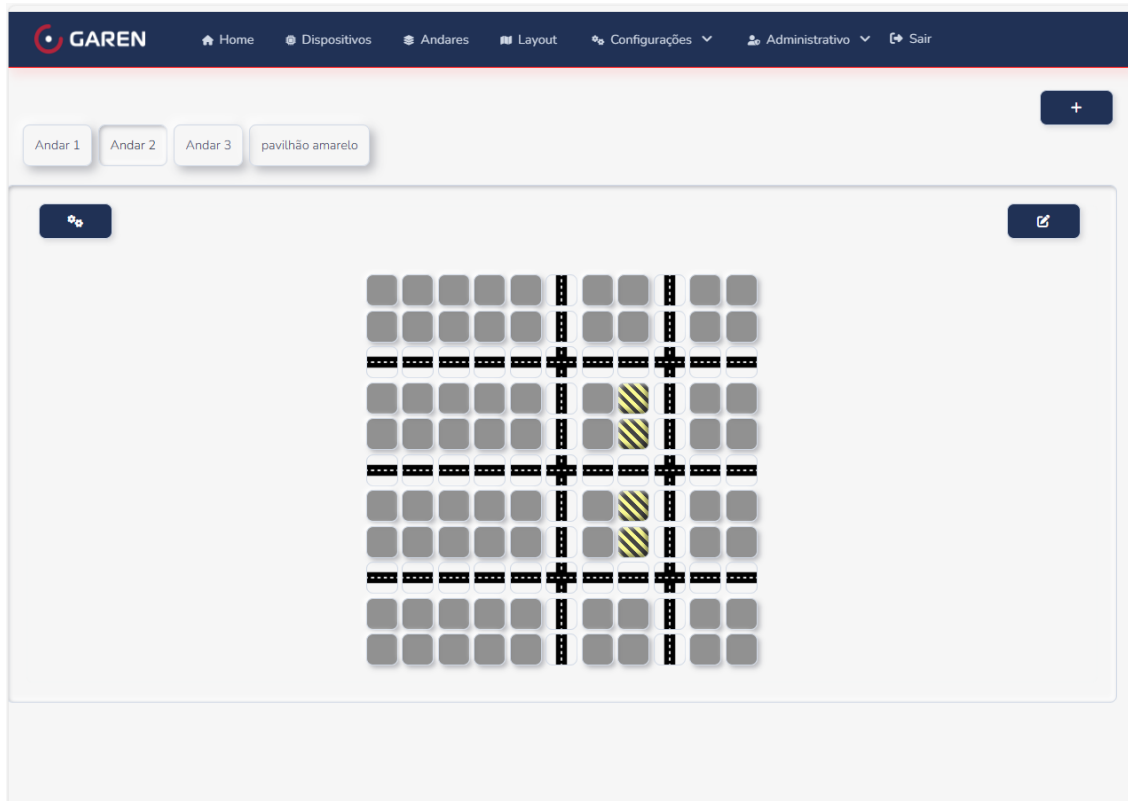
Editando: Andar 1

Andar 1

Cancelar Salvar

*Figura 9: Edição da tela de andares*

## 5.4 Layout dos andares e sincronização



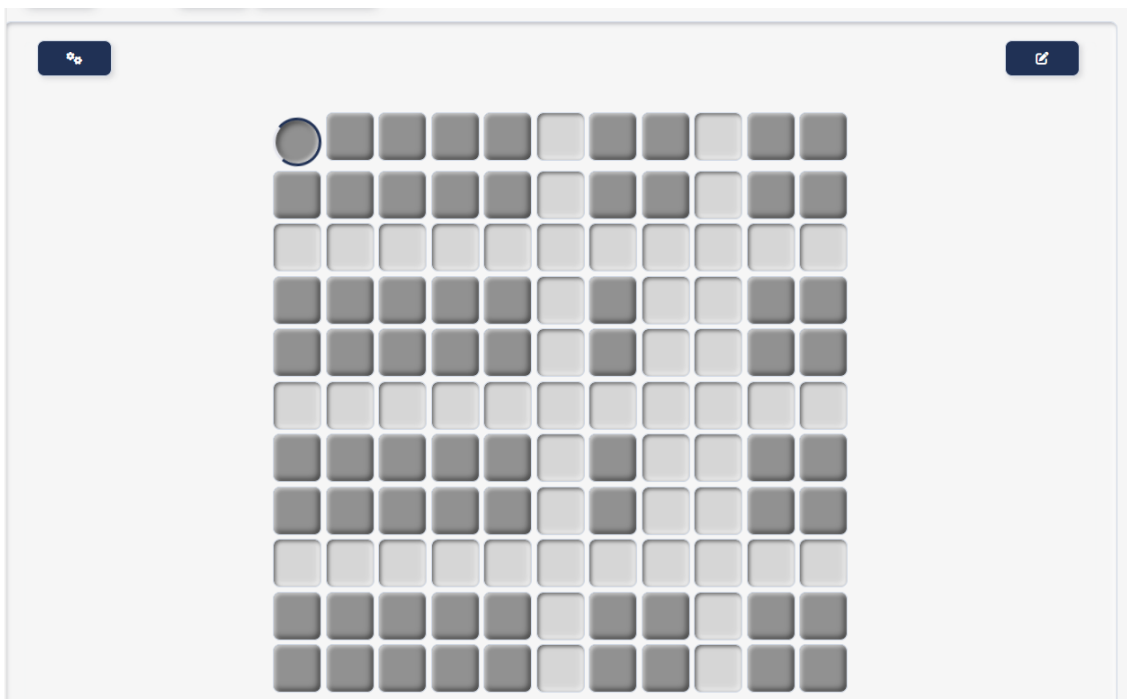
*Figura 10: Tela layout estacionamento*

A tela de layout oferece uma interface gráfica interativa para representar visualmente a estrutura de cada andar do estacionamento. Ela permite ao operador configurar o layout dos andares, visualizar a posição dos dispositivos instalados e realizar a vinculação física entre sensores e vagas por meio de um modo de sincronização.



*Figura 11: Tela Edição de Andar*

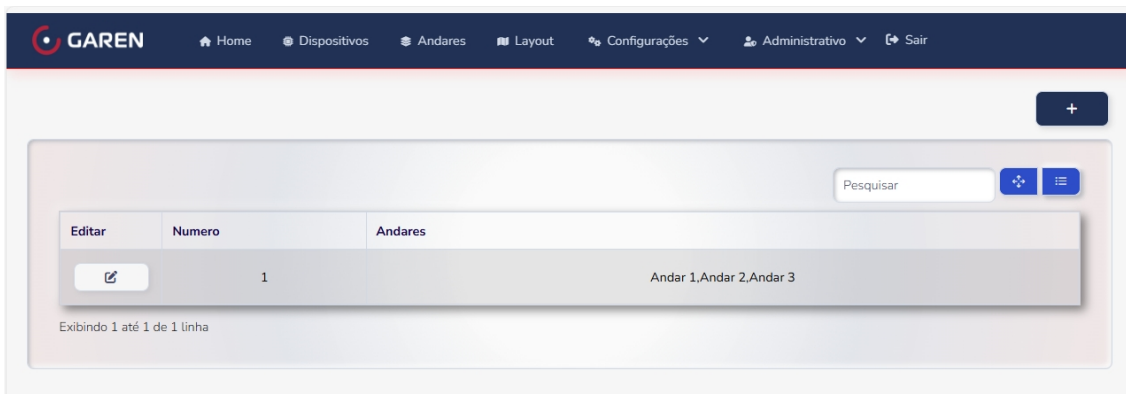
A matriz do estacionamento, assim como seu nome, pode ser alterada na parte na edição.



*Figura 12: Tela de sincronização de dispositivo*

Ao clicar no ícone de engrenagem, a interface entra no modo de sincronização, expandindo visualmente a tela e habilitando a vinculação entre dispositivos físicos e vagas do mapa.

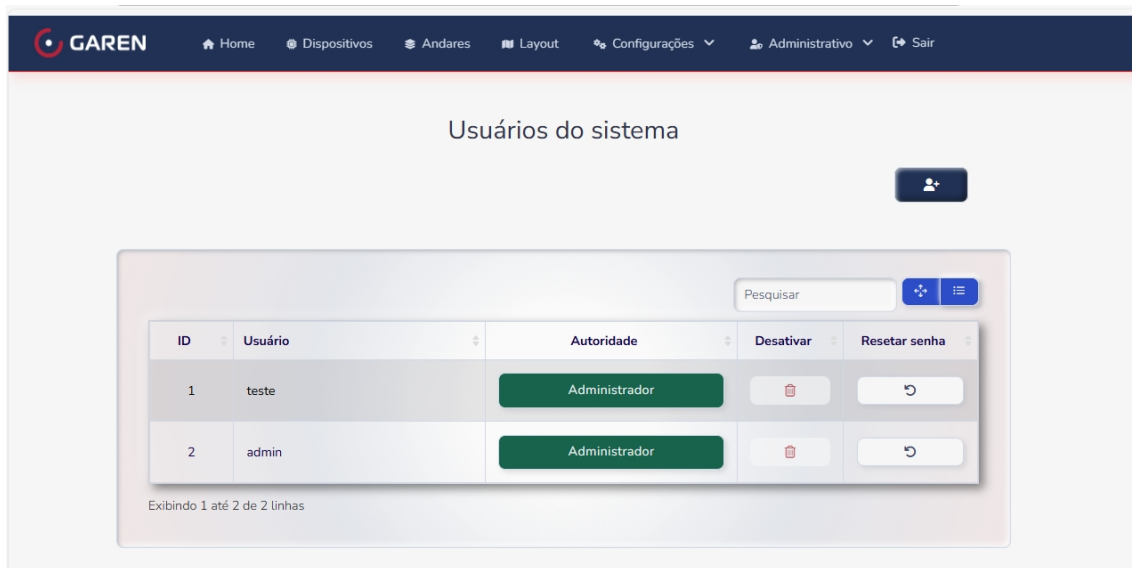
## 5.5 Display



*Figura 13: Tela configuração de display*

A tela de **Configuração de Displays** permite o gerenciamento dos displays de exibição instalados nos andares do estacionamento. Esses displays são responsáveis por apresentar informações aos usuários, como disponibilidade de vagas e futuramente a orientações de navegação. A interface oferece recursos para visualizar, editar e cadastrar novos displays, além de associá-los aos andares correspondentes.

## 5.6 Usuários



*Figura 14: Tela de cadastro de usuario*

Aqui é possível adicionar, desativar e resetar a senha de usuarios cadastrados.

## 6. Código fonte e Comunicação

### 6.1. Estrutura do projeto

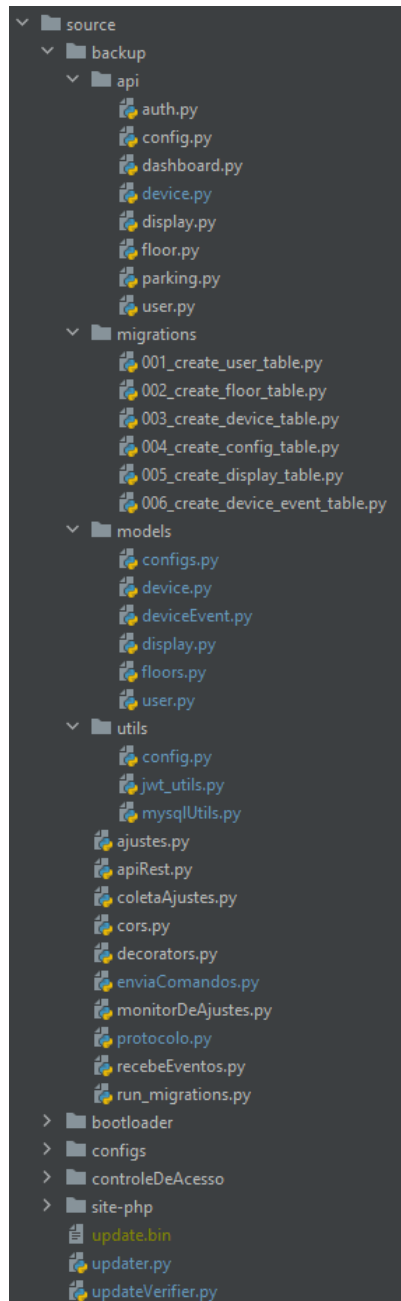


Figura 15: Estrutura de arquivos

## 6.2 Comunicação

O sistema implementa duas camadas de comunicação distintas para garantir a integração completa entre hardware e software, a comunicação com o software é realizada através de uma API REST, enquanto com a parte do hardware, é feita de forma serial.

### 6.2.1 Software de estacionamento.

O software de estacionamento pode ser acessado através do link [192.168.1.100/estacionamento](http://192.168.1.100/estacionamento) (substitua o ip da placa conforme configurado). Por default, o usuario e senha é: **teste**

#### 6.2.1 API REST

A API REST é o núcleo de comunicação entre o software de gerenciamento e os demais componentes do sistema, operando por padrão na porta 5000. Ela oferece funcionalidades essenciais como consulta em tempo real do status das vagas, recuperação de dados históricos e métricas operacionais, configuração de displays e gerenciamento de andares, além da administração de usuários e personalização do layout do estacionamento. Essa interface centralizada garante integração eficiente, controle completo e escalabilidade para todas as operações do sistema.

### 6.2.2 Comunicação Serial

O sistema utiliza duas portas seriais para comunicação direta com os sensores e display. A descrição dos campos pode ser observada no arquivo "Comunicação dispositivo.ods". O arquivo `enviaComandos.py` é responsável pela sincronização de status de facial e display, ele envia de tempos em tempos uma requisição para cada dispositivo conectado na placa controladora para requisitar o seu status atual, fazendo a atualização em seguida do display e banco de dados. O arquivo `recebeEventos.py` por outro lado, fica responsável pelo processamento de alguns eventos gerados de forma ativa, como sincronização, autenticação, e botão ( botao seria usado para sugestão de vagas )

### 6.2.2.1 descrição enviaComandos.py

```
def getDeviceStatus():
    cont = 0
    loggar("Iniciou envia comandos")
    erro_counter = {} # Dicionário para rastrear erros por dispositivo

    while True:
        try:
            default_height = Configs.get_default_height()
            devices = Device.select_all_devices_height_and_id()
            for device in devices:
                device_id = device[0]
                height = device[1] if device[1] is not None else default_height[0]
                status = generate_status(device_id, height)
                result = send_command(status, device[2], device_id)
                if not result:
                    if device_id in erro_counter:
                        erro_counter[device_id] += 1
                    else:
                        erro_counter.setdefault(device_id, 0)

            except Exception as e:
                loggar(f"Erro geral no database_command_verifier: {e}")

            floors = Floors.select_floor()
            cont = 1
            for display in Display.get_all():
                if(display!="0"):
                    floor_ids = display["floors"]
                    total = 0
                    for floor_id in floor_ids:
                        total += Device.select_count_devices(floor_id, "free")
                    print("enviando "+str(total)+" para "+str(display["display_number"]))
                    print(generate_display_response(display["display_number"], total))
                    PortaSerial.write(bytes.fromhex(generate_display_response(display["display_number"], total)))
            mensagem = PortaSerial.read_until(b'\xFF')
            print(mensagem)
            time.sleep(0.005)
```

Figura 16: Método de busca de status de dispositivos

Este arquivo é responsável por fazer a busca de status dos dispositivos, basicamente ele percorre a lista de dispositivos e para cada dispositivo, ele envia uma mensagem de status (0xAA, 0x01, 0x03, 0x09, 0xID\*, 0xID\*, 0xFF) e o dispositivo com aquele ID e retorna um status a qual é salvo no banco de dados, depois é atualizado todos os displays.

### 6.2.2.2 descrição recebeEventos.py

```
def serialListener():
    loggar('Iniciou recebe eventos')
    while True:
        try:
            message = PortaSerial.read_until(b'\xFF')
            if message:
                message = message.hex().upper()
                loggar("Mensagem obtida na serial: " + message)
                result = protocolo.parse_command(message)
                loggar("Mensagem parseada: " + str(result))
                device_response=[]
                if(result["type"] == "auth"):...
                if(result["type"] == "sync"):...
                if(result["type"] == "botao"):...

                if(device_response):
                    loggar("Enviando resposta: " + device_response)
                    PortaSerial.write(bytes.fromhex(device_response))
            time.sleep(1)
        except Exception as e:
            loggar("Erro: " + str(e))
```

Figura 17: Processamento de recebimento de evento

Este arquivo é responsável por o processamento de alguns eventos de dispositivos, como:

- **Autenticação:** quando um novo dispositivo conecta pela primeira vez, ele requisita um ID único dele, esse ID será o enviado em requisições futuras.
- **Sincronização:** Quando sincronizado, o dispositivo recebe um ID já existente, e passa a se comportar como ele.
- **Botão:** O evento de botão será responsável pela indicação de vaga próxima da entrada.



## 7. Endpoints

A documentação swagger dos endpoints da API está disponível via Swagger, acessível em {ip\_da\_placa}:5000, a baixo estará uma breve explicação sobre eles:

### 7.1 Autenticação

```
@auth_ns.route('/login/')
class Login(Resource):
    @auth_ns.expect(login_model)
    @auth_ns.response(200, 'Login realizado com sucesso', token_model)
    @auth_ns.response(401, 'Credenciais inválidas')
    def post(self):
        data = request.get_json()
        username = data.get('username')
        password = data.get('password')
        os.system("busybox date -u " + strftime('%Y.%m.%d-%H:%M:%S', localtime(data.get('epoch'))))
        os.system("busybox hwclock -w")

        user = User.find_by_username(username)
        if user and check_password_hash(user.password_hash, password) and user.active:
            token = encode_jwt({
                'username': user.username,
                'id': user.id,
                'exp': (datetime.datetime.utcnow() + datetime.timedelta(hours=1)).timestamp()
            }, Config.JWT_SECRET_KEY)

            return jsonify({'token': token})

        else:
            return make_response(jsonify({"message": "Invalid credentials"}), 401)
```

*Figura 18: Rota de login*

Autentica um usuário com base nas credenciais fornecidas e retorna um token JWT em caso de sucesso. É também utilizado para sincronizar o horário junto ao login.

## 7.2 Configuração

```
@config_ns.route('/')
class Config(Resource):
    @config_ns.response(200, 'Configurações atuais retornadas com sucesso', config_output_model)
    @token_required
    def get(self, current_user_id):
        return make_response(jsonify({'height': Configs.get_default_height_and_width()[0],
                                      'width': Configs.get_default_height_and_width()[1]}), 200)

    @config_ns.expect(config_input_model)
    @config_ns.response(200, 'Configurações atualizadas com sucesso')
    @token_required
    def post(self, current_user_id):
        data = request.get_json()
        sensor_height = data.get('height')
        sensor_width = data.get('width')
        print(sensor_height)
        print(sensor_width)
        Configs.set_default_height_and_width(sensor_height, sensor_width)
        return make_response(jsonify(), 200)
```

*Figura 19: Rotas de configuração*

Aqui é possível buscar e alterar a altura e largura padrão que os sensores deverão respeitar para a detecção de automóveis.

## 7.3 Dashboard

```
@dashboard_ns.route('/')
class Dashboard(Resource):
    @token_required
    @dashboard_ns.doc(params={'floor_id': 'ID do andar para consulta'})
    @dashboard_ns.response(200, 'Dados do painel retornados com sucesso', dashboard_output_model)
    @dashboard_ns.response(400, 'Parâmetro floor_id ausente')
    def get(self, current_user_id):
        params = request.args
        floor_id = params.get("floor_id")

        if not floor_id:
            return make_response(jsonify({'error': 'floor_id is required'}), 400)

        device_counts = Device.get_device_counts_by_floor(floor_id)

        free_devices = device_counts.get('free', 0)
        occupied_devices = device_counts.get('occupied', 0)
        offline_devices = device_counts.get('offline', 0)
        total = free_devices + occupied_devices + offline_devices

        return make_response(jsonify({
            'free': free_devices,
            'occupied': occupied_devices,
            'total': total,
            'problem': offline_devices
        }), 200)
```

*Figura 20: Rota dos cards no dashboard*

Aqui mostra um resumo dos dispositivos de um andar específico. Você manda o ID do andar, e ela te devolve quantos estão livres, ocupados, com problema e o total.

```
@dashboard_ns.route('/table/')
class DashboardTable(Resource):
    @token_required
    @dashboard_ns.doc(params={'floor_id': 'ID do andar para consulta'})
    @dashboard_ns.response(200, 'Lista de dispositivos retornada com sucesso', [device_table_item])
    @dashboard_ns.response(400, 'Parâmetro floor_id ausente ou inválido')
    def get(self, current_user_id):
        params = request.args
        floor_id = params["floor_id"]
        response = []
        for device in Device.select_devices_without_blocked_and_roads(floor_id):
            if(device!="0"):
                response.append({'id':device[0],"name":device[1],"status":device[2],"signal":device[3],
                                "x":device[4],"y":device[5],"sensor_height":device[6]})
        return make_response(jsonify(response), 200)
```

*Figura 21: Rota de tabelas dashboard*

Aqui a tabela principal do dashboard busca a lista com os dispositivos de um andar específico, mostrando dados como nome, status, sinal, posição e altura do sensor. Só traz os que não estão bloqueados e não são vias. Precisa mandar o id do estacionamento.

```

@dashboard_ns.route('/parking_hours/')
class ParkingHours(Resource):
    @token_required
    @dashboard_ns.response(200, 'Horas de estacionamento calculadas com sucesso', [parking_hours_item])
    @dashboard_ns.response(400, 'Parâmetros obrigatórios ausentes', error_model)
    @dashboard_ns.response(500, 'Erro interno ou timeout', error_model)
    def get(self, current_user_id):
        floor_id = request.args.get("floor_id")
        start_time = request.args.get("start_time")
        end_time = request.args.get("end_time")

        if not all([floor_id, start_time, end_time]):
            return make_response(jsonify({
                "error": "Parâmetros obrigatórios: floor_id, start_time, end_time"
            }), 400)

        try:
            devices = Device.select_devices_without_blocked_and_roads(floor_id)
            if not devices:
                return make_response(jsonify([]), 200)

            device_ids = [d[0] for d in devices]

            parking_hours = DeviceEvent.calculate_parking_hours_parallel_batches(
                device_ids, start_time, end_time, max_workers=2
            )
            response = [
                {
                    "device_id": device[0],
                    "name": device[1],
                    "parking_hours": parking_hours.get(device[0], 0)
                }
                for device in devices
            ]
            return make_response(jsonify(response), 200)

        except Exception as e:
            print(f"[ERROR] ParkingHours: {e}")
            return make_response(jsonify({"error": "Timeout ou erro interno"}), 500)

```

*Figura 22: Rota de calculo de quantidade de horas gastas em vagas de um período.*

Essa rota calcula quanto tempo cada dispositivo ficou ocupado em um determinado andar, dentro de um intervalo de tempo que você manda. Se der erro ou demorar demais, ela devolve um erro genérico.

```

@dashboard_ns.route('/occupancy_durations/')
class OccupancyDurations(Resource):
    @token_required
    @dashboard_ns.doc(
        params={
            'floor_id': 'ID do andar',
            'start_time': 'Início do intervalo',
            'end_time': 'Fim do intervalo'
        }
    )
    @dashboard_ns.response(200, 'Durações de ocupação retornadas com sucesso', [occupancy_duration_item])
    @dashboard_ns.response(400, 'Parâmetros inválidos ou ausentes', error_model)
    def get(self, current_user_id):
        params = request.args
        floor_id = params.get("floor_id")
        start_time = params.get("start_time")
        end_time = params.get("end_time")

        if not floor_id or not start_time or not end_time:
            return make_response(jsonify({"error": "parâmetros inválidos"}), 400)

        devices = Device.select_devices_without_blocked_and_roads(floor_id)
        device_ids = [d[0] for d in devices]
        if not device_ids:
            return make_response(jsonify([]), 200)

        buckets = DeviceEvent.count_occupancy_durations_bulk(
            device_ids,
            int(start_time),
            int(end_time)
        )

        return make_response(jsonify(buckets), 200)

```

*Figura 23: Rota de calculo de duração média de ocupação de vaga*

Essa rota calcula quanto tempo os dispositivos ficaram ocupados dentro de um intervalo de tempo. Ela agrupa essas durações em faixas (tipo menos de 5 minutos, entre 15 e 60, mais de 6 horas, etc.).

```

@dashboard_ns.route('/parking_events/')
class ParkingEvents(Resource):
    @token_required
    @dashboard_ns.doc(
        params={
            'floor_id': 'ID do andar',
            'start_time': 'Início do intervalo',
            'end_time': 'Fim do intervalo'
        }
    )
    @dashboard_ns.response(200, 'Eventos de estacionamento retornados com sucesso', [parking_event_item])
    @dashboard_ns.response(400, 'Parâmetros inválidos ou ausentes', error_model)
    def get(self, current_user_id):
        params = request.args
        floor_id = params.get("floor_id")
        start_time = params.get("start_time")
        end_time = params.get("end_time")

        if not floor_id or not start_time or not end_time:
            return make_response(jsonify({"error": "parâmetros inválidos"}), 400)

        devices = Device.select_devices_without_blocked_and_roads(floor_id)
        device_ids = [d[0] for d in devices]

        if not device_ids:
            return make_response(jsonify({"error": "Nenhum dispositivo encontrado"}), 400)

        start_ms = int(start_time)
        end_ms = int(end_time)
        eventos_por_device = DeviceEvent.count_connection_events_bulk(device_ids, start_ms, end_ms)

        response = [
            {
                "device_id": device[0],
                "name": device[1],
                "connection_count": eventos_por_device.get(device[0], 0)
            }
            for device in devices
        ]

        return make_response(jsonify(response), 200)

```

*Figura 24: Rota de quantidade de ocupação de vagas em um período de tempo*

Esse end-point conta quantas vezes cada dispositivo foi ocupado ou ficou offline dentro de um intervalo de tempo.

```

@dashboard_ns.route('/vacancies/')
class VacancieEvents(Resource):
    @token_required
    @dashboard_ns.doc(
        params={'floor_id': 'ID do andar (opcional, será usado o primeiro se não informado)'}
    )
    @dashboard_ns.response(200, 'Contagem de dispositivos retornada com sucesso', vacancy_output_model)
    def get(self, current_user_id):
        params = request.args
        floor_id = params.get("floor_id")

        if not floor_id:
            floor_id = Floors.select_floor()[0][0]

        devices_vacancies = Device.select_count_devices_triple(floor_id, "occupied", "offline", "free")
        print(devices_vacancies)
        return make_response(dict(devices_vacancies), 200)

```

*Figura 25: Rota de contagem de dispositivo por status*

Essa rota retorna a quantidade de dispositivos por status, ou seja, quantos dispositivos estão ocupados, livres ou offlines em um andar.

```

@dashboard_ns.route('/volume/')
class VolumeEvents(Resource):
    @token_required
    @dashboard_ns.doc(
        params={
            'floor_id': 'ID do andar (opcional, será usado o primeiro se não informado)',
            'start_time': 'Início do intervalo (timestamp em ms)',
            'end_time': 'Fim do intervalo (timestamp em ms)'
        }
    )
    @dashboard_ns.response(200, 'Volume de eventos retornado com sucesso', [volume_item])
    @dashboard_ns.response(400, 'Parâmetros ausentes ou inválidos', error_model)
    def get(self, current_user_id):
        params = request.args
        floor_id = params.get("floor_id")
        start_time = params.get("start_time")
        end_time = params.get("end_time")
        if not floor_id:
            floors = Floors.select_floor()
            if not floors or floors == "0":
                return make_response(jsonify({"error": "Nenhum andar encontrado"}), 400)
            floor_id = floors[0][0]

        devices = Device.select_devices_without_blocked_and_roads(floor_id)
        device_ids = [d[0] for d in devices]

        if not device_ids:
            return make_response(jsonify({"error": "Nenhum dispositivo encontrado"}), 400)

        data = DeviceEvent.count_volume_by_epoch(device_ids, start_ms, end_ms)
        return make_response(jsonify(data), 200)

```

*Figura 26: Rota de contagem de volume de carros*

Aqui é calculado o volume de eventos de ocupação e offline dos dispositivos ao longo do tempo, dividindo o intervalo em blocos de 15 minutos.



## 7.4 Display

```
@display_ns.route('/')
class floor(Resource):
    @token_required
    @display_ns.expect(display_input_model)
    @display_ns.response(200, 'Display criado com sucesso', display_response_model)
    @display_ns.response(404, 'Display já existe', display_conflict_model)
    def post(self, current_user_id):
        data = request.get_json()
        display_number = data.get('display_number')
        floors = data.get('floors')
        display = Display.find_by_number(display_number)
        if display:
            return make_response(jsonify({'status': 'display_number_already_exists'}), 404)

        _id = Display.create_config(display_number, ",".join(floors))
        return make_response({'id': _id}, 200)

    @token_required
    @display_ns.expect(display_delete_model)
    @display_ns.response(200, 'Display removido com sucesso')
    def delete(self, current_user_id):
        data = request.get_json()
        _id = data.get('id')
        Display.delete_config(_id);
        return make_response(jsonify({}), 200)

    @token_required
    @display_ns.expect(display_update_model)
    @display_ns.response(200, 'Display atualizado com sucesso')
    def put(self, current_user_id):
        data = request.get_json()
        _id = data.get('id')
        display_number = data.get('display_number')
        floors = data.get('floors')
        print(_id)
        print(display_number)
        print(floors)
        Display.update_config(_id, display_number, ",".join(floors))
        return make_response(jsonify({}), 200)
```

Figura 27: Rotas de configuração de display

Essa rota serve pra criar, atualizar ou remover um display no sistema. No POST, você cadastra um novo display com número e andares. No DELETE, remove um pelo ID. No PUT, atualiza os dados de um display existente.

```

@display_ns.route('/all/')
class allDisplays(Resource):
    @token_required
    @display_ns.response(200, 'Lista de displays retornada com sucesso', [display_all_model])
    def get(self, current_user_id):
        response = []
        for display in Display.get_all():
            if(display!="0"):
                floor_ids = display["floors"]
                floor_names = []
                for floor_id in floor_ids:
                    print(floor_id)
                    floor_data = Floors.select_floor_by_id(floor_id)
                    if floor_data!="0":
                        print("floor_data")
                        print(floor_data)
                        floor_names.append(floor_data[0][1])
                response.append({
                    "id": display["id"],
                    "number": display["display_number"],
                    "floors": display["floors"],
                    "floor_names": floor_names
                })

        return make_response(jsonify(response), 200)

```

*Figura 28: Rota para busca de displays cadastrados*

Essa rota retorna todos os displays cadastrados, junto com os andares que cada um está ligado e os nomes desses andares.

## 7.4 Andares

```
@floor_ns.route('/')
class Floor(Resource):
    @token_required
    @floor_ns.response(200, 'Lista de andares retornada com sucesso', [floor_model])
    def get(self, current_user_id):
        response = []
        for floor in Floors.select_floor():
            if(floor!="0"):
                response.append({"id":floor[0],"name":floor[1],"width":floor[2],"height":floor[3],"x_in":floor[4],"y_in":floor[5]})
        return make_response(jsonify(response), 200)

    @token_required
    @floor_ns.response(200, 'Andar criado com sucesso', model=floor_ns.model('FloorCreateResponse', {
        'id': fields.String(description='ID do novo andar')
    }))
    def post(self, current_user_id):
        if(Floors.select_count_floors()>9):
            return
        _id = Floors.add_floor();
        Floors.update_floor_name("Andar "+str(_id),_id);
        return make_response({"id":_id}, 200)

    @token_required
    @floor_ns.expect(floor_delete_model)
    @floor_ns.response(200, 'Andar removido com sucesso')
    def delete(self, current_user_id):
        data = request.get_json()
        _id = data.get('id')
        Floors.delete_floor(_id);
        return make_response(jsonify({}), 200)

    @token_required
    @floor_ns.expect(floor_update_model)
    @floor_ns.response(200, 'Andar atualizado com sucesso')
    def put(self, current_user_id):
        data = request.get_json()
        _id = data.get('id')
        width = data.get('width')
        height = data.get('height')
        name = data.get('name')
        if name:
            Floors.update_floor_name(name,_id);
        if width and height:
            Floors.update_floor(_id,width,height)
        return make_response(jsonify({}), 200)
```

Figura 29: Rotas de configuração de andares

Essa rota permite listar, criar, atualizar ou remover andares. No GET, ela devolve todos os andares cadastrados. No POST, cria um novo. No DELETE, remove um andar pelo ID. No PUT, atualiza nome, largura e altura.

```

@floor_ns.route('/all/')
class updateDevice(Resource):
    @token_required
    @floor_ns.response(200, 'Status dos andares retornados com sucesso', [floor_status_model])
    def get(self, current_user_id):
        response = []
        for floor in Floors.select_floor():
            if(floor!="0"):
                floor_id = floor[0]
                occupied_devices = Device.select_count_devices(floor_id,status='occupied')
                free_devices = Device.select_count_devices(floor_id,status='free')
                offline_devices = Device.select_count_devices(floor_id,status='offline')
                total = offline_devices+free_devices+occupied_devices;
                response.append({"id":floor_id,"name":floor[1],"free":free_devices,"occupied":occupied_devices,"total":total,"problem":offline_devices,"width":floor[2],"height":floor[3]})

        return make_response(jsonify(response), 200)

```

*Figura 30: Rota para busca de informação de todos os andares*

Essa rota traz o status geral de todos os andares: quantos dispositivos estão livres, ocupados, com problema e o total. Também inclui as dimensões de cada andar.

## 7.5 Vagas

```

@parking_ns.route('/')
class parking(Resource):
    @token_required
    @parking_ns.doc(params={'floor_id': 'ID do andar'})
    @parking_ns.response(200, 'Lista de dispositivos retornada com sucesso', [device_model])
    def get(self, current_user_id):
        params = request.args
        floor_id = params["floor_id"]
        response = []
        for device in Device.select_devices(floor_id):
            if(device!="0"):
                response.append({"id":device[0],"name":device[1],"status":device[2],"signal":device[3],"x":device[4],"y":device[5]})
        return make_response(jsonify(response), 200)

```

*Figura 31: Rota de listagem de dispositivos*

Essa rota lista todos os dispositivos de um andar específico, mostrando dados como nome, status, sinal e posição.

```

@parking_ns.route('/block/')
class parking(Resource):
    @token_required
    @parking_ns.doc(params={
        'floor_id': 'ID do andar',
        'x': 'Coordenada X',
        'y': 'Coordenada Y'
    })
    @parking_ns.response(200, 'Dispositivo atualizado ou criado com sucesso')
    def post(self, current_user_id):
        params = request.args
        floor_id = params["floor_id"]
        x = params["x"]
        y = params["y"]
        response = []
        device = Device.select_device(floor_id,x,y)[0];
        if(device!="0"):
            print(device)
            status =device[2];
            if(status=="block"):
                Device.update_device(device[0],"offline")
            elif(status=="road"):
                Device.update_device(device[0],"block")
            else:
                Device.update_device(device[0],"road")

        else:
            Device.create_device(str(x)+"-"+str(y),"road",floor_id,x,y)

        return make_response(jsonify(response), 200)

```

*Figura 32: Rota de configuração de tipo de dispositivo*

Essa rota altera o tipo de um dispositivo com base na posição (x, y) em um andar. Se o dispositivo já existe, ela muda o status em sequência: de "road"(via) pra "block"(obstaculo), de "block" pra "offline", e de qualquer outro pra "road". Se não existir, ela cria um novo com status "road".