PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO

Dashboard Web para visualização de dados coletados de dispositivos IoT

Théo Falcato Ribeiro Palmeirim de Athayde

PROJETO FINAL DE GRADUAÇÃO

CENTRO TÉCNICO CIENTÍFICO - CTC

DEPARTAMENTO DE INFORMÁTICA

Curso de Graduação em Ciência da Computação

Rio de Janeiro, Novembro, 2024

Théo Falcato Ribeiro Palmeirim de Athayde

Dashboard Web para visualização de dados coletados de dispositivos IoT

Relatório de Projeto Final, apresentado ao curso de Ciência da Computação da PUC-Rio como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação.

Orientador: Anderson Oliveira da Silva

Rio de Janeiro

Novembro de 2024

Resumo

Falcato Ribeiro Palmeirim de Athayde, Théo. Oliveira da Silva, Anderson. Dashboard Web para visualização de dados coletados de dispositivos IoT. Rio de Janeiro, 2024. Número de páginas p. Relatório Final de Projeto Final – Centro Técnico Científico, Departamento de Informática. Pontifícia Universidade Católica do Rio de Janeiro.

O trabalho envolve criar um dashboard web para visualizar dados coletados de dispositivos IoT em tempo real através de gráficos. A aplicação permite ao usuário cadastrar novos sensores e começar a inserir no banco de dados as informações enviadas por esses sensores através de um protocolo de mensagens nomeado MQTT (Message Queing Telemetry Transport).

Palavras-chave

Dashboard, desenvolvimento Web, MQTT, CRUD, Tempo real

Abstract

Falcato Ribeiro Palmeirim de Athayde, Théo. Oliveira da Silva, Anderson. Web Dashboard to visualize data collected from IoT devices. Rio de Janeiro, 2024. Number of pages p. Relatório Final de Projeto Final – Centro Técnico Científico, Departamento de Informática. Pontifícia Universidade Católica do Rio de Janeiro.

This work is about creating a web dashboard to visualize data collected from IoT devices in real time. The application allows the user to register new sensors and start to insert the informations sent from these sensor in the database through a message protocol named MQTT (Message Queing Telemetry Transport).

Keywords

Dashboard, Web development, MQTT, CRUD, Real time.

Sumário

Sumário imagens

1. Introdução

O projeto foi o desenvolvimento de uma aplicação web para visualizar dados recebidos de dispositivos IoT em tempo real. Essa aplicação envolve a criação de dashboards e paineis de visualização que podem ser associados a um sensor de quaisquer dispositivos que o usuário deseje. Esses dados são recebidos através do protocolo de mensagens MQTT e armazenados em um banco de dados relacional MySQL.

O GIST é um laboratório da PUC focado em sistemas distribuídos e soluções relacionadas a IoT. A principal motivação da aplicação é dar liberdade para membros do GIST ou outros interessados a ter um acesso organizado, rápido e seguro aos dados dos dispositivos interessados.

Ter acesso a esses dados pode além de ser útil para acompanhar o andamento das funções de um dispositivo, como também ajudar na detecção de informações críticas, como um sensor de frequência cardíaca que aponta um ritmo fora dos parâmetros ideais, que pode até sinalizar algo mais grave. A aplicação permite checar esses limites e avisar ao usuário caso um dado novo esteja abaixo ou acima desses números.

A aplicação foi inteiramente desenvolvida em uma máquina virtual com o sistema operacional Linux CentOS 7. O projeto utilizou o framework Django[1] (versão 3.2), um framework Web que possui uma estrutura Model View Template e utiliza as linguagens Python (versão 3.6.8) com conceitos de Orientação a Objetos, além de HTML, CSS[9] e Javascript. Além disso utilizou-se o Mosquitto (versão 1.8) para o MQTT e um banco de dados MySQL (versão 8.1).

Essa relatório está organizado da seguinte forma: a seção 2 aborda as aplicações estado da arte que inspiraram esse projeto; a seção 3 apresenta os ambientes do sistema de dashboard, o ambiente de visualização dos dados via dashboard (aplicação web), o ambiente de publicação de tópicos através do serviço MQTT e o ambiente de coleta de tópicos para o banco de dados e execução dessa coleta em background através do systemd; a seção 4 apresenta a arquitetura da aplicação web, especificando os componentes do framework Django; a seção 5 apresenta a organização do armazenamento dos dados, especificando os modelos do banco de dados, cada uma das colunas das tabelas e as relações entre as entidades; a seção 6 explica o funcionamento do protocolo MQTT e sua arquitetura, além de como a coleta dos dados funciona na aplicação; a seção 7 apresenta o método de criação de paineis para a visualização dos dados através de gráficos; a seção 8 fala mais sobre os dashboards e a customização da organização das visualizações; a seção 9 aborda a autenticação dos usuários e as permissões de cada grupo (staff e usuário regular); a seção 10 apresenta a técnica AJAX, que permite atualizar elementos da página sem necessariamente recarregá-la; a seção 11 fala sobre a interface de registro de novos sensores e tópicos; a seção 12 relata um experimento de registro de um novo sensor e tópicos, criação de paineis associado a esses tópicos e a visualização desses paineis em um dashboard; a seção 13 aborda a interface de monitoramento dos sensores, especificando os serviços systemd e o script bash para configuração desses serviços; a seção 14

apresenta os métodos utilizados para realizar o deploy da aplicação; a seção 15 apresenta a conclusão; e, por fim, a seção 16 apresenta trabalhos futuros.

2. Estado da Arte

A aplicação foi inspirada em outras ferramentas web de criação de dashboards e visualização de dados, como Grafana e ThingSpeak. **Grafana**[10] é uma plataforma de código aberto amplamente utilizada para criar dashboards interativos e realizar monitoramento em tempo real. É conhecida pela flexibilidade, integração com diversas fontes de dados e visualizações avançadas. **ThingSpeak**[11] é uma plataforma baseada na nuvem focada em IoT (Internet of Things), ideal para coletar, armazenar e visualizar dados de dispositivos conectados. Ele foi projetado para facilitar a análise de dados de sensores e atuadores.

O objetivo de desenvolver uma ferramenta de dados no estilo de dashboards própria, juntando elementos desses dois outros softwares, visa exercitar o desenvolvimento de uma aplicação Web com o framework Django e AJAX, além de explorar a integração de tópicos publicados no MQTT com a visualização de dados na aplicação e assegurar a integração do dashboard com outros projetos de pesquisa do GIST Lab. Muitas funcionalidades interessantes existem nesse Estado da Arte, e a ideia é o projeto atender as necessidades e gerar valor para os usuários como nestes softwares.

Outro ponto importante da aplicação é que ela será usada futuramente no projeto de monitoramento de pacientes e, por causa da privacidade e segurança dos dados da saúde, o dashboard deverá ser capaz de trabalhar com dados criptografados, que em geral não é suportado por outras aplicações. Ter um dashboard próprio permite esse caso de uso.

[**Figura 1**] Figura representativa do Grafana.

3. Configuração do Ambiente

O sistema de dashboard é composto por vários ambientes.

3.1 Ambiente de visualização de dados via dashboard (aplicação web)

O ambiente de visualização dos dados foi implementado utilizando o framework Django[1], que se baseia primariamente na linguagem Python. O Django também fornece “Templates”, que funcionam como o frontend da aplicação, e permite o uso de linguagens web como CSS, HTML e Javascript. Para começar o desenvolvimento foi necessário atualizar a versão do Python e instalar o Django.

Após instalar o Django alguns passos iniciais são necessários para estruturar a aplicação. O Django é organizado em dois níveis principais:

- Projeto: O projeto é o nível mais alto da configuração da aplicação, onde ficam as configurações globais, como o arquivo **settings.py**. Ele pode ser criado com o comando “django-admin startproject **myproject**” onde myproject é o nome dado ao projeto principal.
- App: É um módulo independente dentro do projeto, com uma funcionalidade específica. No caso deste projeto, foi criado um app chamado **dashboard**. Para criar um app utilizamos o comando “python manage.py startapp **myapp**” onde myapp é o nome atribuído ao app.

Por padrão o Django utiliza um banco de dados SQLite, mas como estamos utilizando um banco MySQL precisamos alterar essa configuração. Para isso foi necessário instalar o conector mysqlclient. Também é necessário alterar as credenciais no arquivo de configuração do projeto Django (o settings.py). Por fim, necessitamos rodar os comandos “python manage.py makemigrations” e “python manage.py migrate” para inicializar o banco de dados.

Podemos então rodar o projeto localmente utilizando “python manage.py runserver”.

3.2 Ambiente de publicação de tópicos (serviço MQTT)

É por meio do ambiente de publicação de tópicos que conseguimos receber os dados coletados dos dispositivos IoT. MQTT é um protocolo de mensagens extremamente leve, ideal para conexão com dispositivos remotos. Para realizar essa conexão utilizamos a biblioteca de python **paho-mqtt**. Foi necessário instalar essa biblioteca e configurar opções como as credenciais do broker. Após isso utilizamos um comando do Django para iniciar a conexão e começar a realizar o subscribe dos dados. Cada execução do comando começa a receber dados de tópicos relacionados a um sensor, e os dispositivos publicam esses dados através dos tópicos.

3.3 Ambiente de coleta de tópicos para banco de dados (systemd)

O comando Django permite iniciar a conexão do broker MQTT, e os dados recebidos passam a ser adicionados no banco de dados. Porém é necessário iniciar múltiplas conexões, uma para cada sensor, e também é necessário manter essas conexões rodando em background. Para isso foi utilizado uma funcionalidade do sistema Linux, rodar serviços systemd. O serviço foi configurado para rodar os comandos Django de maneira genérica, podendo facilmente ser iniciado ou parado.

4. Arquitetura da Aplicação Web

A arquitetura principal da aplicação é baseada no framework **Django**[1], que é robusto, de alto nível, e amplamente utilizado para o desenvolvimento de aplicações web escaláveis, seguras e de fácil manutenção. O Django adota o princípio **DRY (Don't Repeat Yourself)**[4], que promove a reutilização de código e simplifica a manutenção ao evitar redundâncias.

O Django organiza sua arquitetura seguindo o padrão **MVT (Model View Template)**. Esse padrão divide a aplicação em três camadas principais, promovendo a separação de responsabilidades:

- **Models (Modelos)**: Os **Models** são responsáveis por manipular os dados e representam as tabelas do banco de dados no código da aplicação. O Django utiliza um poderoso sistema de **ORM (Object-Relational Mapping)**, que abstrai a necessidade de escrever SQL diretamente. Isso permite interagir com o banco de dados usando objetos Python, o que torna o código mais legível e fácil de manter.
- **Views (Visualizações)**: As **Views** processam as requisições dos usuários e funcionam como uma ponte entre os **Models** e os **Templates**. Elas recuperam os dados do banco de dados (via Models), processam a lógica necessária e enviam os dados para os Templates, que serão exibidos ao usuário.
- **Templates**: Os **Templates** são os arquivos HTML que compõem a interface visual da aplicação. Eles são responsáveis por renderizar os dados enviados pelas Views, utilizando a **linguagem de templates do Django**, que permite incluir lógica básica, como loops e condicionais. Os Templates permitem a reutilização de estruturas HTML através de herança, tornando o código mais modular.

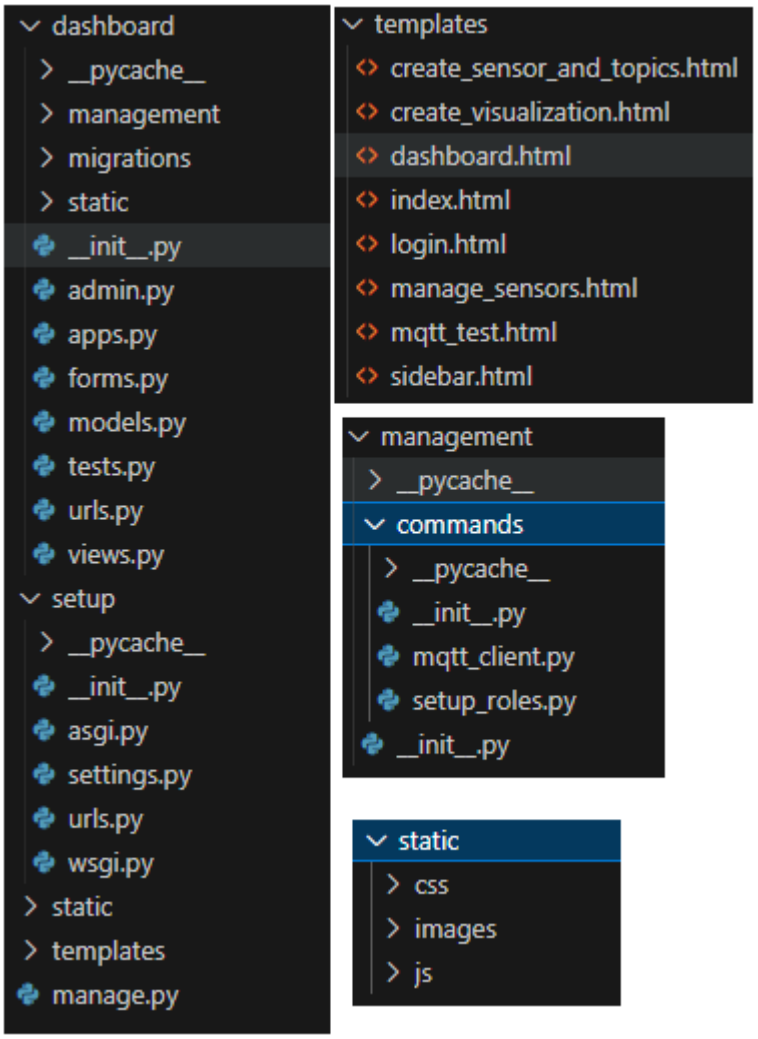
A arquitetura Django também possui uma organização de arquivos que facilita o gerenciamento do código com pastas para os Templates, Models, Views, Urls (rotas específicas da aplicação), além das configurações globais do projeto.

No projeto temos a pasta do app "dashboard" com arquivos de configuração mais gerais (**admin.py** e **apps.py**), além do **forms.py** que organiza todos os formulários utilizados na aplicação, o **models.py** com todos os modelos, o **views.py** que concatena as views, o **urls.py** com as rotas referentes ao app dashboard e o **tests.py** é utilizado para armazenar os testes. Na pasta "management" temos os comandos customizados, o **mqtt_client.py** que inicia a conexão com o broker MQTT e **setup_roles.py** cria os usuários e os grupos com permissão ao ser executado.

A pasta setup possui as configurações do projeto, também possui os arquivos de configuração mais gerais (**asgi.py** e **wsgi.py**), além do arquivo **settings.py** com as configurações principais e o arquivo **urls.py** com as rotas principais da aplicação.

A pasta template reúne os templates da aplicação, que como especificado acima são os arquivos html que correspondem às interfaces do projeto.

Por fim a pasta “static” reúne os arquivos estáticos, como arquivos de CSS, imagens ou arquivos Javascript.



[**Figura 2**] Configuração dos diretórios do projeto.

5. Armazenamento e Estrutura dos Dados

A aplicação possui alguns modelos, que são basicamente as tabelas contidas no banco de dados. Existem 5 modelos:

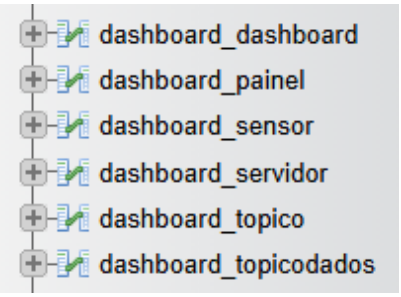
Sensor: o sensor é a estrutura principal dos dispositivos, cada dispositivo pode ter um sensor com vários tópicos. Esse modelo possui um nome e credenciais. Essas credenciais (usuário e senha) são necessárias para realizar a conexão MQTT. Além disso, possui um ID que funciona como chave primária, que é gerado pelo próprio Django.

Tópico: um tópico está sempre relacionado a um sensor, o tópico é quem vai realmente estar associado a um dado de um dispositivo. O tópico possui um nome, que é sempre no estilo “nome_do_sensor”/”nome_do_topico”. Por exemplo: sensor1/temperatura, no caso esse tópico está associado ao “sensor1” e mostra temperaturas. Temos também um limite inferior e um limite superior, esses limites indicam quando um dado está fora dos parâmetros, se esse for o caso será indicado no gráfico e o usuário será notificado. Também possui um data_type, que indica o tipo do dado que vem desse tópico. Ele pode ser (float, integer, string, boolean ou JSON). O modelo também possui uma coluna que representa o ID do sensor que esse tópico está relacionado, uma chave estrangeira para sensor.

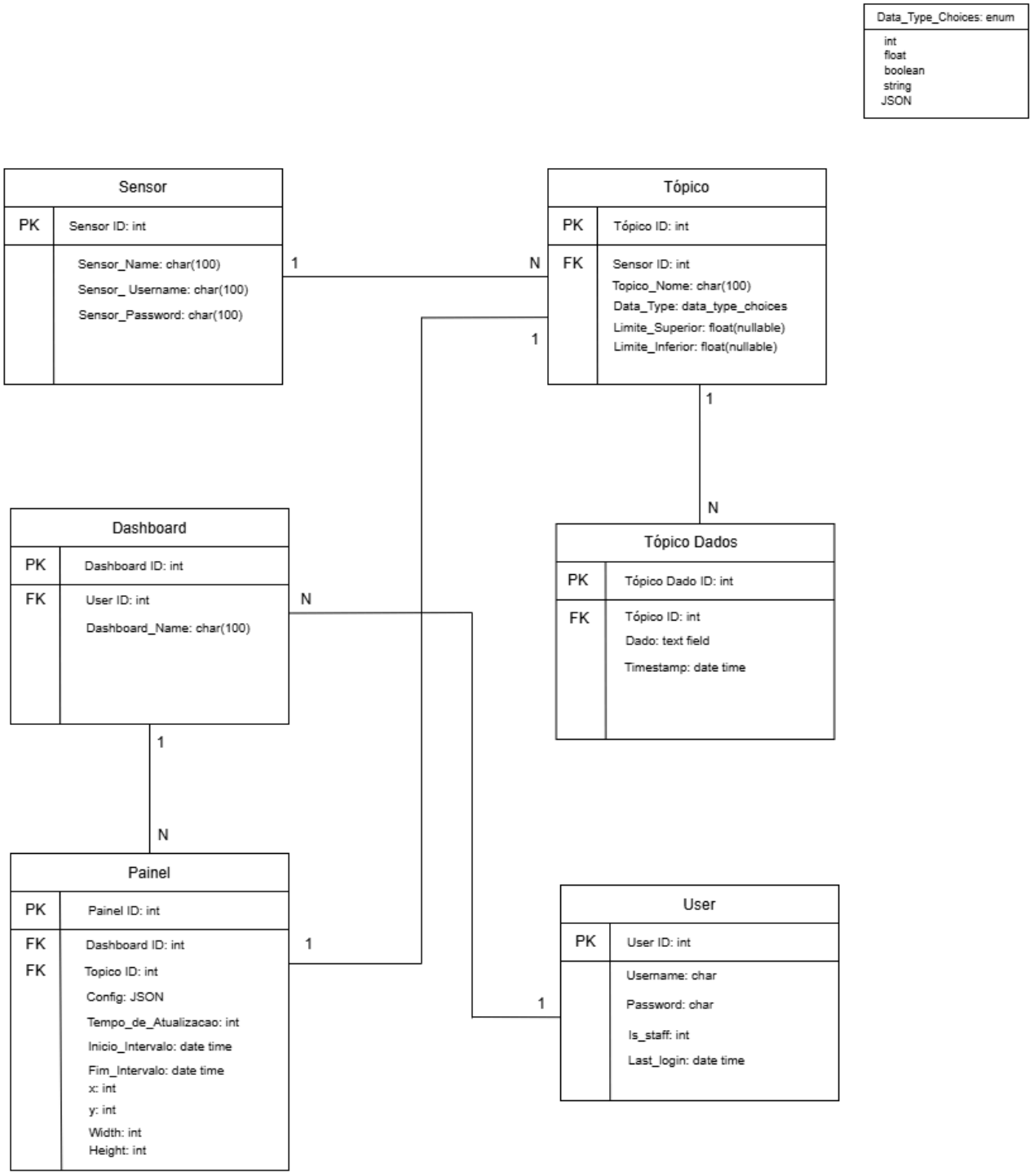
Tópico dados: essa é a tabela que armazena os dados coletados dos dispositivos IoT. Ela possui a coluna com o valor do dado em si, além de uma coluna com o Timestamp de quando esse dado foi recebido. Também possui o ID do tópico, chave estrangeira para tópico.

Painel: O painel é a tabela responsável por armazenar os dados da visualização em si. Ele possui o tempo de atualização do recebimento de dados painel em si (polling), Possui a informação da timestamp do início do intervalo dos dados e o timestamp do final do intervalo de visualização daquele painel. Também tem o id do dashboard e do tópico que esse Painel está relacionado (chaves estrangeiras). Também armazena o tamanho width e height do painel, além da posição x e y no dashboard. E por último outros parâmetros como cor, ou tipo de gráfico (barra, linha ou scatter).

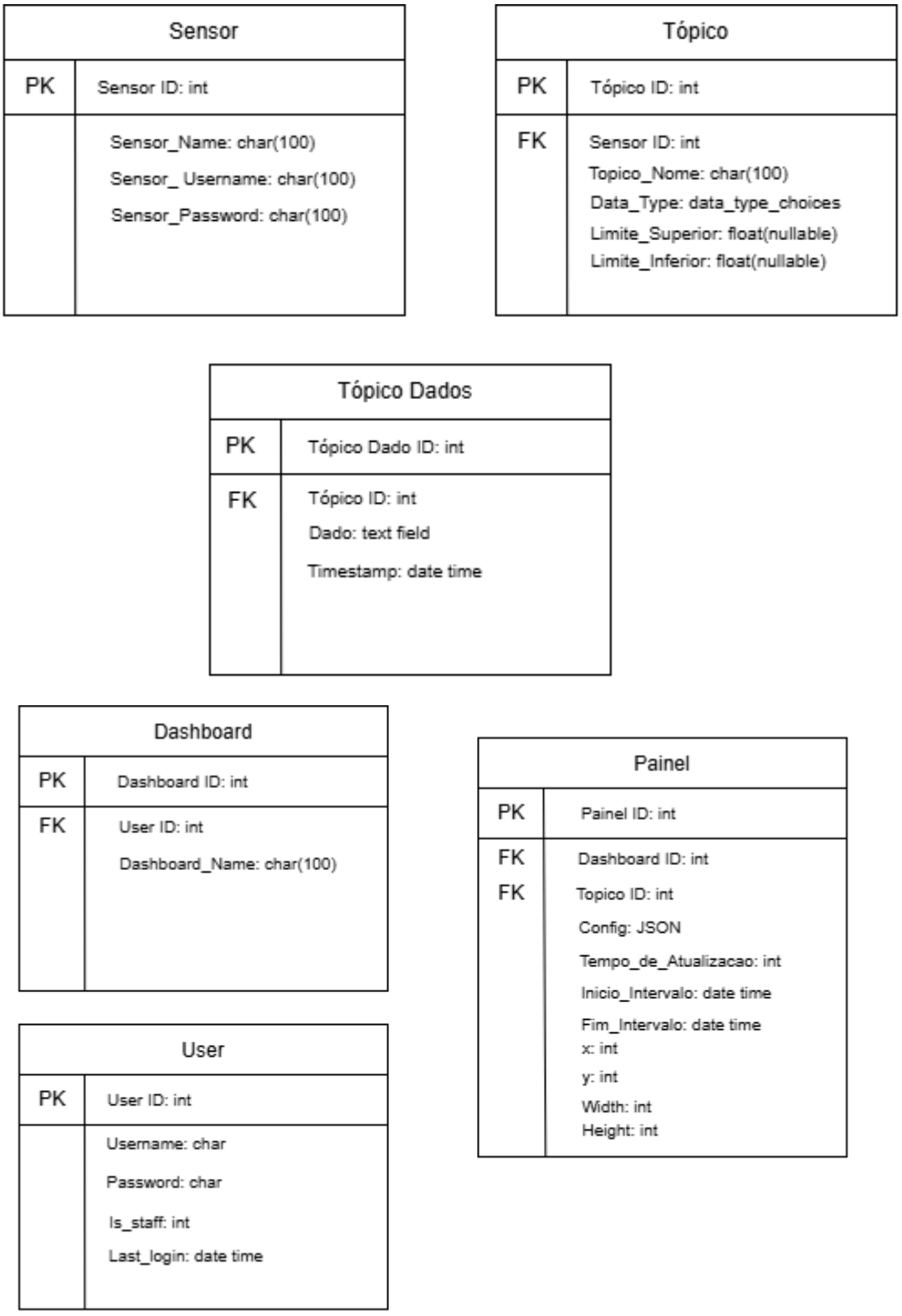
Dashboard: o dashboard representa o local onde os paineis estão concatenados. Ele armazena seu nome, o usuário a quem está associado, além de seu ID gerado automaticamente.



[**Figura 3**] Tabelas do banco de dados.



[**Figura 4**] Modelo lógico do Banco de Dados.



[**Figura 5**] Diagrama dos modelos ampliado.

6. MQTT e Coleta de Dados

A coleta dos dados do dispositivo IoT se dá através do protocolo de mensagens MQTT[12], em particular utilizando a biblioteca paho.mqtt[7] do python e o mosquitto. O MQTT é leve, eficaz e tem um baixo consumo de energia. Ele funciona através de um modelo publish/subscribe, onde o broker fica escutando os dados publicados pelos dispositivos.

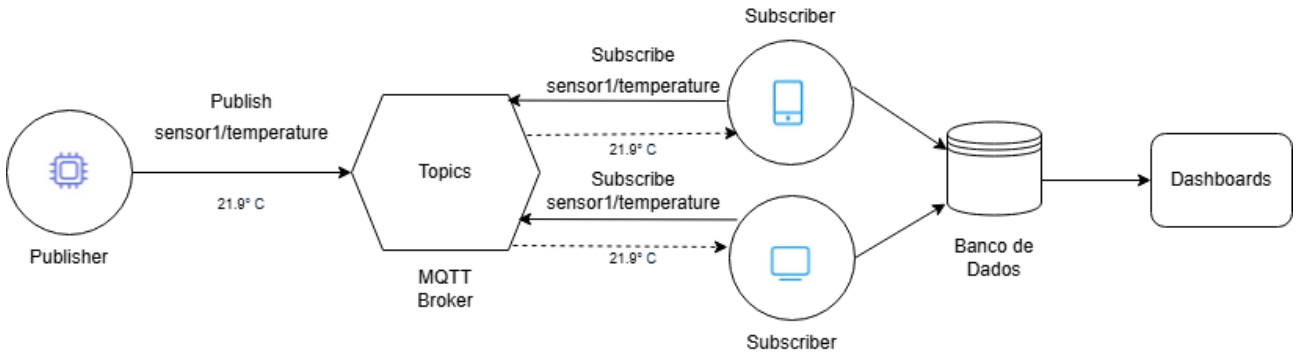
A figura abaixo exemplifica essa arquitetura. O publisher é responsável por publicar mensagens para um tópico. Ele só pode mandar dados para um tópico por vez.

Os subscribers recebem as mensagens se inscrevendo para tópicos e podem se inscrever em múltiplos tópicos ao mesmo tempo.

O broker é responsável por receber as mensagens dos publisher e direcioná-las aos subscribers. Ele também pode lidar com requisições como conectar, desconectar, se inscrever e se desinscrever. O sistema usa apenas um broker.

As rotas de mensagens MQTT são baseadas em tópicos, na aplicação o nome dos tópicos se organiza como: "*nome_do_sensor/nome_do_tópico*". O comando django para realizar a conexão com o broker permite o subscriber receber mensagens de todos os tópicos relacionados a um sensor utilizando wildcards.

Na aplicação após passar pelos subcribers os dados são inseridos no banco de dados e podem ser visualizados futuramente nos dashboards.



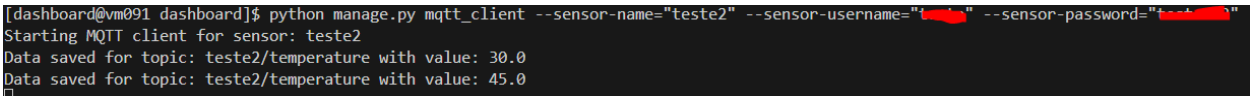
[**Figura 6**] Arquitetura MQTT no projeto.

Para realizar esse monitoramento na aplicação, foi utilizado um recurso muito interessante do Django, a criação de um comando novo. Esse comando foi criado na pasta de "commands" do app, e pode ser ativado através da linha de comando. No comando é passado o nome do sensor que vai ser escutado, e as credenciais desse sensor. Se o comando for executado com sucesso, todos os dados dos tópicos relacionados a esse sensor serão recebidos e armazenados no banco.

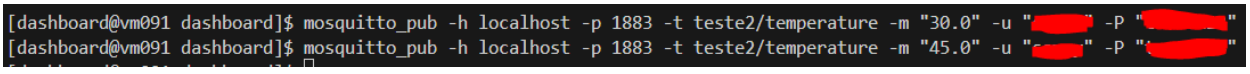
Para deixar ainda mais eficiente, foi criado um serviço systemd para rodar o comando de monitoramento em background. Esse serviço pode ser utilizado para subir uma instância do comando Django recebendo os dados enquanto estiver ativa. Como mencionado anteriormente o comando de monitoramento necessita das credenciais do sensor, portanto para conseguir subir o serviço armazenamos as credenciais em um arquivo de environment e o serviço lê esse arquivo.

Além disso, foi criada uma interface no frontend da aplicação que permite ao usuário começar o serviço, checar seu status, parar e criar as credenciais caso não existam. Para o usuário conseguir fazer todas essas ações foi criado um script bash com todas as opções possíveis e esse arquivo recebeu acesso sudo. Utilizando então a função subprocess do Django o usuário tem acesso a esse script bash de maneira visual.

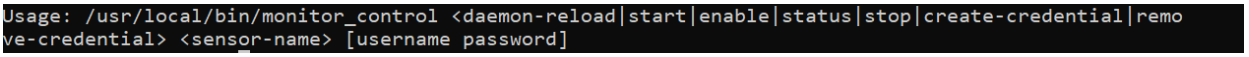
Ao utilizar o comando Django personalizado para iniciar o broker MQTT, a conexão é estabelecida para todos os tópicos dos sensores de um dispositivo e todos os dados recebidos são adicionados ao banco de dados. Para iniciar a conexão é feito através da linha de comando utilizando o nome do sensor e suas credenciais: “python manage.py mqtt_client –sensor name “*nome_do_sensor*” –sensor_username “*usuario_do_sensor*” –sensor_password “*senha_do_sensor*”. As mensagens podem ser transmitidas pelos dispositivos usando o mosquitto através do comando: “mosquitto_pub -h “*hostname*” -p “*port-number*” -t “*nome_do_topico*” -m “*dado_recebido*” -u “*username_broker*” -P “*broker password*”.



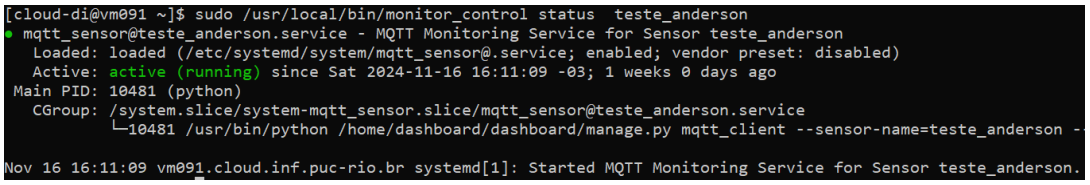
[**Figura 7**] Broker MQTT iniciado.



[**Figura 8**] Publicação de mensagens MQTT.



[**Figura 9**] Script bash que inicia os serviços systemd para um sensor.



[**Figura 10**] Status do serviço relacionado a um sensor específico.

7. Criação e Personalização de Paineis e Visualizações

A visualização dos dados se dá através de gráficos. Esses gráficos podem ser adicionados a paineis, e podem ser customizados de diversas maneiras. Primeiramente, para criar um gráfico associado a um painel o usuário tem acesso a um template, uma página html voltada para a personalização e construção do gráfico em si. Nessa página o usuário tem acesso a um formulário, ao preencher esse formulário com as informações necessárias pode ser gerado uma prévia de um gráfico, que pode então ser adicionado ao dashboard. Essa interface de criação de paineis pode ser acessada a partir de qualquer dashboard. Ao entrar na interface é salvo a informação de qual dashboard essa página foi acessada, portanto caso o painel seja salvo ele vai ser adicionado ao dashboard do contexto atual.

O formulário de criação de gráficos permite selecionar o tópico a qual o painel estará associado, o intervalo de data de quais dados devem ser mostrados, além do tempo de atualização dos paineis (os dados novos são adicionados aos paineis em tempo real). Também é possível configurar opções visuais como o tipo de gráfico (barra, linha ou scatter) e a cor dos dados.

Para a geração e visualização de gráficos foi utilizada a biblioteca open-source Echarts[2]. Ela permite uma grande flexibilidade para a geração de gráficos, além de diversas funcionalidades que são fáceis de serem implementadas.

[**Figura 11**] Formulário de criação de gráficos.

[**Figura 12**] Preview do gráfico que será adicionado ao dashboard.

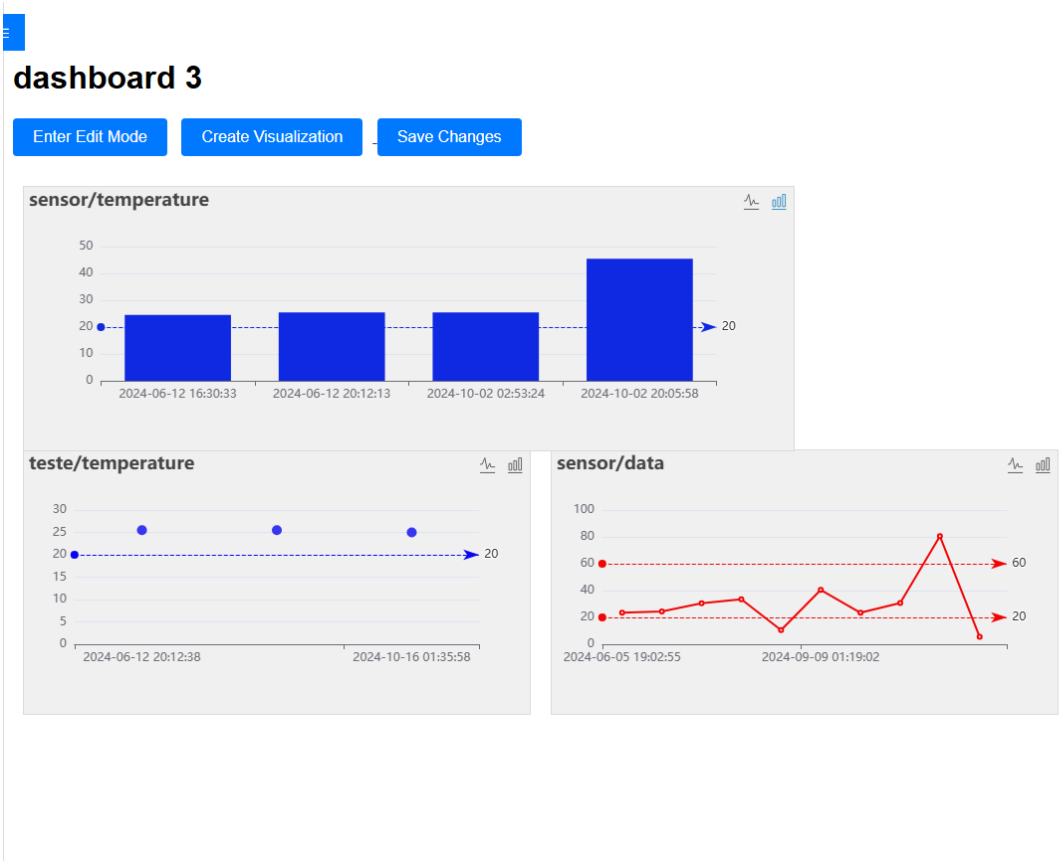
8. Dashboards e manutenção do conteúdo

Os dashboards são a forma principal do usuário acompanhar os dados coletados em tempo real. No dashboard podem existir quaisquer número de paineis, cada um podendo estar associado a um tópico diferente ou com intervalos de amostragem ou atualização dos dados diferentes.

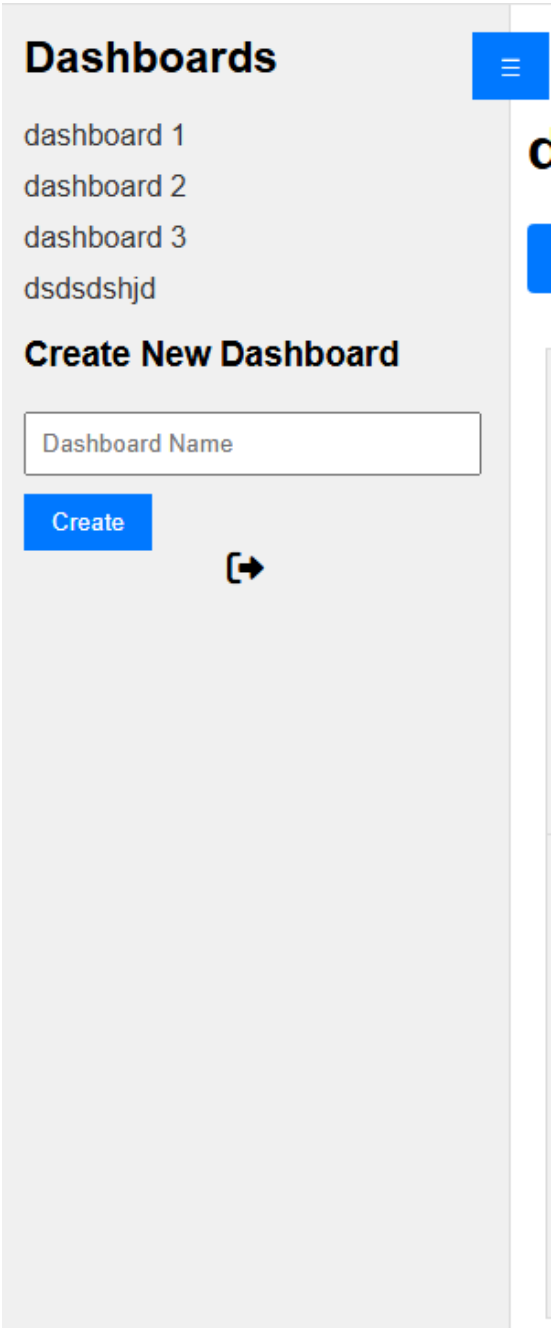
Dashboards podem entrar em modo edição, quando o dashboard está em modo de edição o usuário pode redimensionar o painel ou mudar sua posição. Clicando no botão de salvar essas mudanças são salvas no banco e a configuração se mantém. No modo de edição também é possível deletar os paineis através de um ícone de lixeira que aparece no canto superior direito do painel.

Ao lado do dashboard podemos abrir a sidebar. Na sidebar o usuário pode criar novos dashboards, deletar dashboards, mudar o nome de dashboards e acessar outros dashboards, cada um com seus paineis e configurações.

A biblioteca utilizada para os dashboard foi a Gridstack.js[3], que é uma biblioteca open-source que permite um layout responsivo, além de funcionalidades como drag and drop, resize e fácil acesso a criação e deleção dos paineis.



[**Figura 13**] Dashboard com paineis adicionados.



[**Figura 14**] Sidebar que permite criar e acessar diferentes dashboards, além de permitir a realização do logout.

9. Autenticação de usuários

O framework Django possui um sistema de autenticação pronto para uso, que permite utilizar usuários, grupos e permissões para controlar o acesso a aplicação e suas funcionalidades. O núcleo do sistema de autenticação são os dois modelos, User e Group.

Para armazenar a senha do login de um usuário o Django usa o algoritmo PBKDF2 (Password-Based Key Derivation Function 2) com a hash SHA256, que incorpora SALT e múltiplas iterações.

O modelo User é responsável por armazenar os usuários, com seus usuários e senhas por padrão. O modelo Group serve para agrupar esses usuários e eles vão ser associados a suas permissões.

Na aplicação existem dois grupos, Staff e Regular Users. Os usuários Staff tem maior poder administrativo, eles podem acessar e utilizar as interfaces de criação de Tópicos e Sensores e fazer o monitoramento desses sensores através da interface de monitoramento. Usuários Staff também tem permissões parar criar ou deletar novos usuários.

Usuários regulares têm acesso a criação de dashboards e paineis, e tem permissão de visualizar todos os dados dos tópicos dos dispositivos que o usuário Staff cadastrou e recebeu via MQTT.

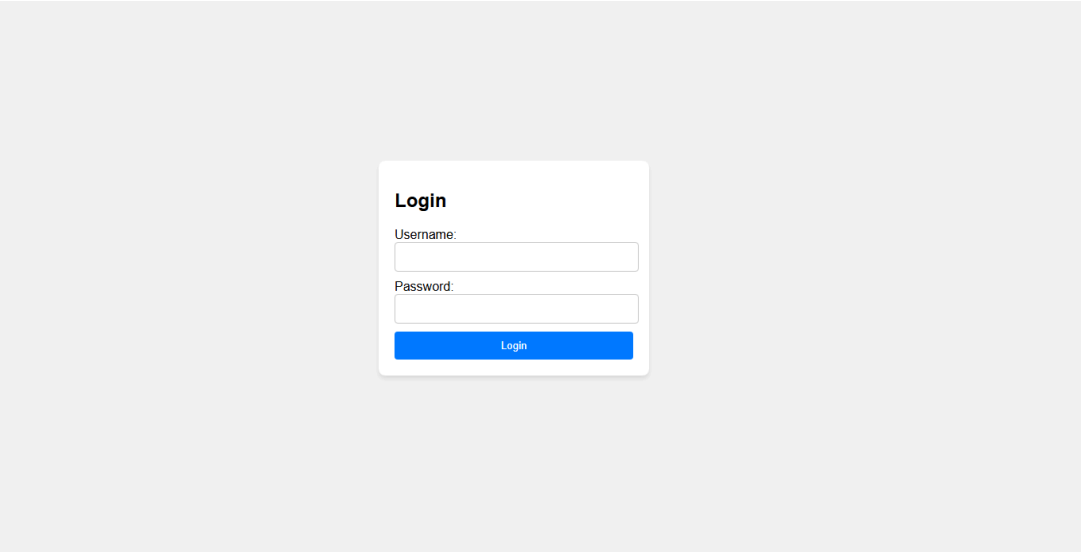
Também existe uma página de login, a aplicação só pode ser utilizada por um usuário logado, seja Staff ou Regular. Para isso, todas as “Views” do projeto recebem um decorator “@login_required”, que é a forma do Django de bloquear o acesso a essas Views. As views que são bloqueadas para usuários que não são staff também recebem o decorator “@permission_required (“permissão aqui”).

A implantação de um sistema de autenticação em duas etapas Time-based One-Time Password (TOTP) com Google Authenticator ou Microsoft Authenticator aumenta o nível de segurança da tela de login da aplicação Web. Esse sistema será implementado futuramente.

[**Figura 15**] Imagem do banco dos grupos de permissão de usuários.

```python
# Add permissions to Staff group
staff_permissions = [
    'add_sensor', 'change_sensor', 'delete_sensor',
    'add_user', 'change_user', 'delete_user',
]
```

[**Figura 16**] Permissões do usuário staff.



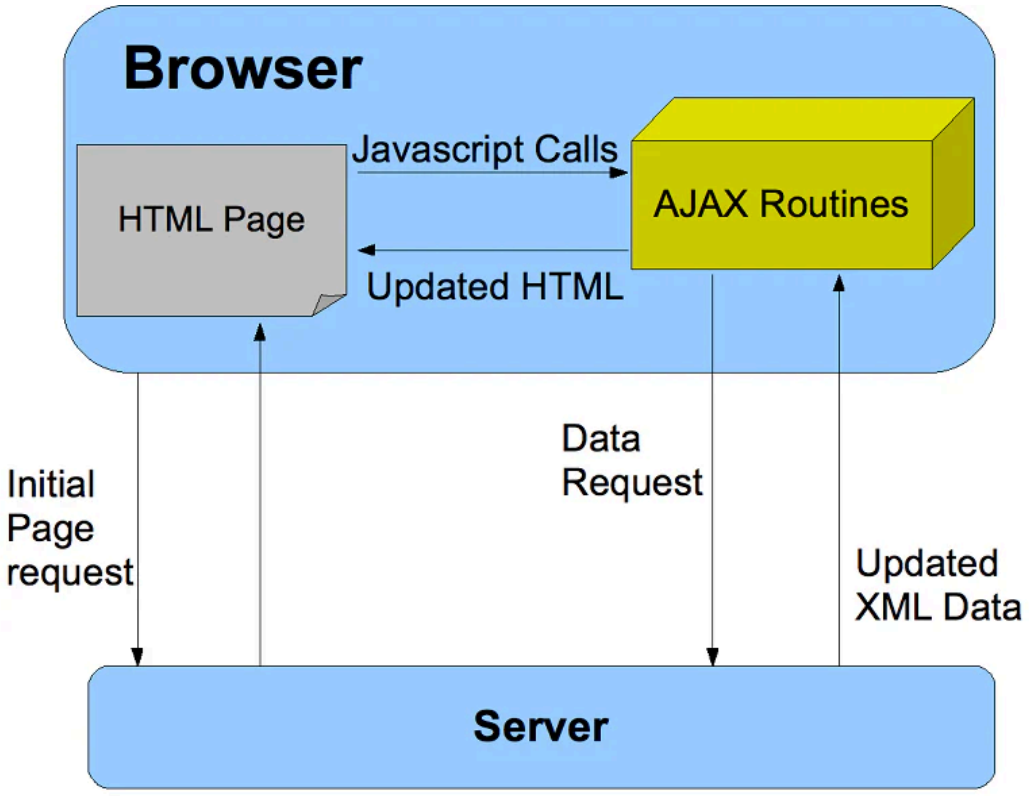
[**Figura 17**] Tela de login da aplicação.

10. AJAX e funcionalidades relacionadas

AJAX (Asynchronous JavaScript and XML) é uma técnica que permite a atualização de partes específicas de uma página web sem a necessidade de recarregar o conteúdo completo. Essa comunicação assíncrona entre o cliente (navegador) e o servidor proporciona uma experiência mais dinâmica e responsiva para o usuário, otimizando o desempenho e reduzindo o tráfego de dados.

No projeto, **AJAX** desempenha um papel fundamental em diversas funcionalidades, tornando o sistema mais interativo e eficiente.

O AJAX é utilizado em algumas partes do projeto, como no Polling dos dados, permitindo a atualização dos dados dos paineis em tempo real. Ele é utilizado na deleção de paineis, para que não seja necessário atualizar a página para removê-lo. Também é utilizado na geração do gráfico no formulário, mostrando o preview do gráfico de forma mais fluida e eficiente.

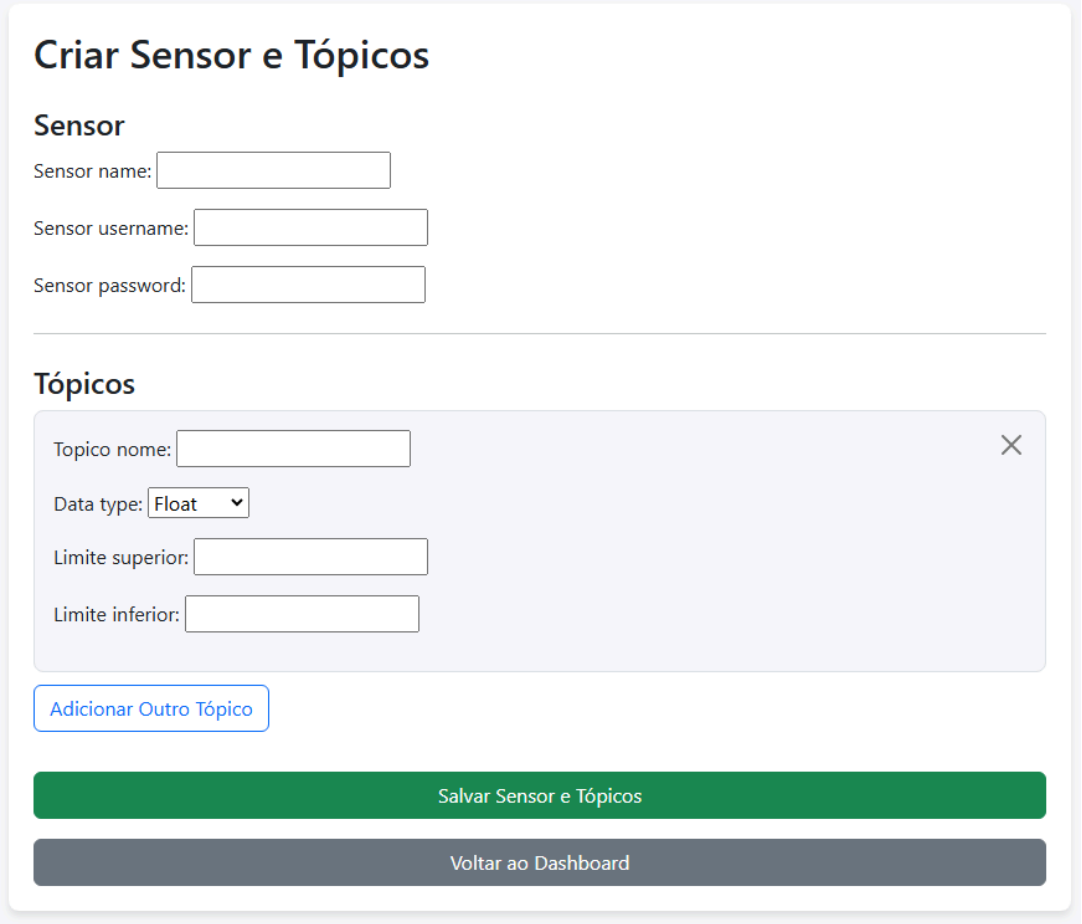


[**Figura 18**] Demonstração da funcionalidade do AJAX.

11. Registro de novos sensores

Os dados recebidos via MQTT estão atrelados a tópicos que por sua vez estão associados a um sensor. A aplicação permite o monitoramento desses tópicos, e permite também o registro de novos sensores e tópicos.

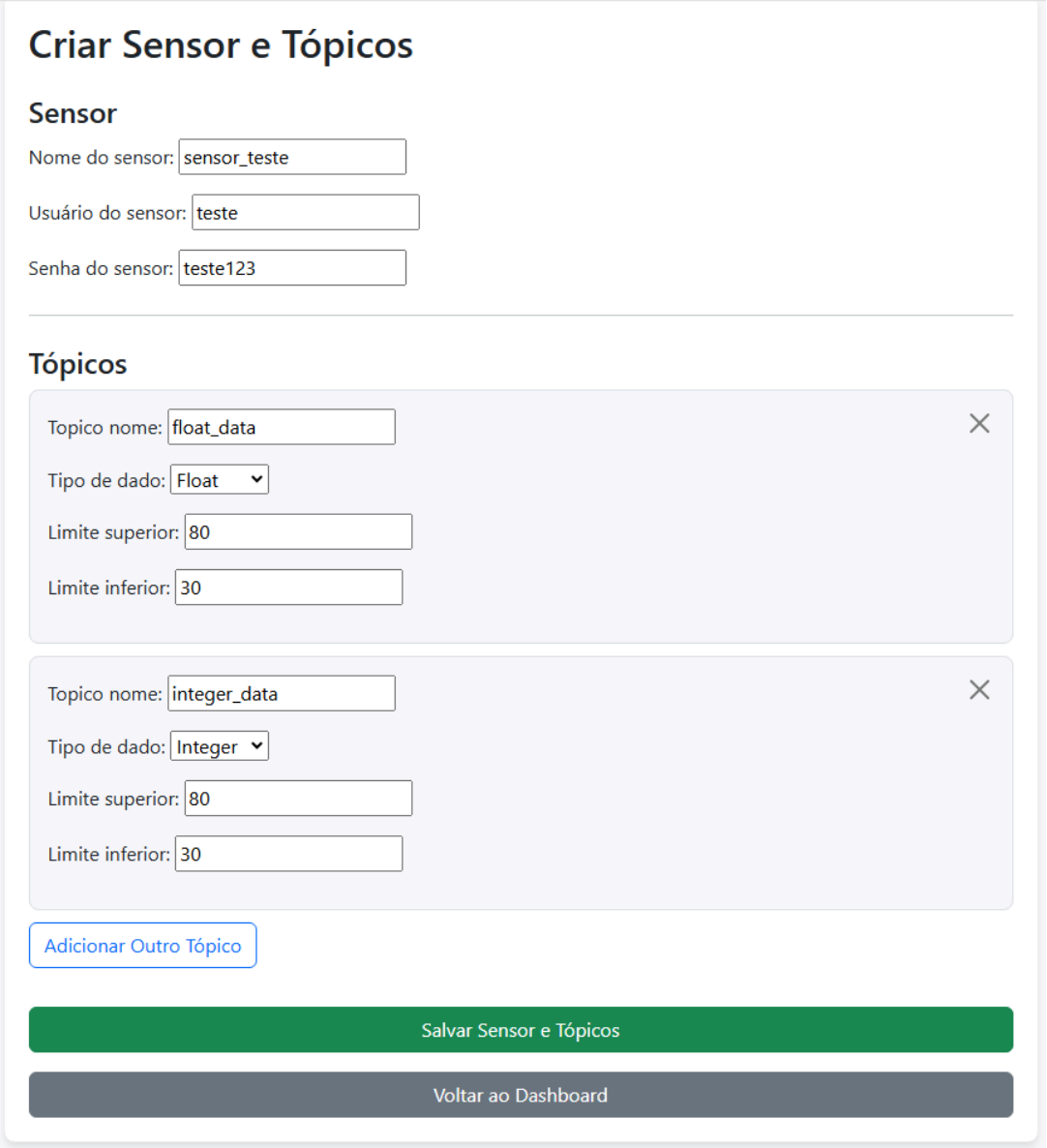
A interface de criação ou registro de novos tópicos tem seu acesso limitado ao usuário staff. Através dessa interface e de um formulário o usuário pode criar um novo sensor com suas credenciais e registrar quaisquer número de tópicos associados a esse sensor. Cada tópico tem o tipo de dado que ele fornece e o limite superior e inferior dos parâmetros desses dados. Após esse cadastro o banco é atualizado e salva o sensor, suas credenciais, os novos tópicos com seus atributos e a chave estrangeira que relaciona o tópico ao sensor.



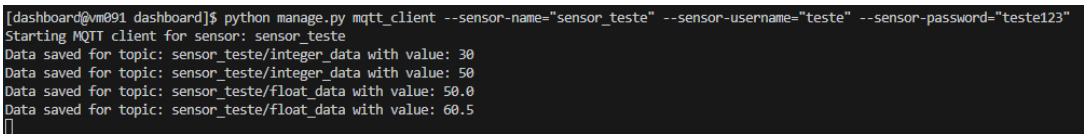
[**Figura 19**] Formulário de registro de sensores e seus tópicos.

12. Experimentos

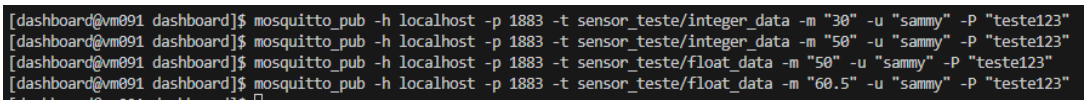
Como experimento foi realizado um teste com dois tópicos, um que publica valores inteiros e outro valores float. Esses dois tópicos estão relacionados a um mesmo sensor. Para esse experimento criamos o sensor e os tópicos através da interface de registro de sensores que o usuário staff tem acesso. Rodamos então o comando Django de conexão ao broker MQTT com as credenciais do sensor e publicamos os dados através do mosquitto_pub. Esses dados são adicionados ao banco e podemos criar os gráficos relacionados a esses tópicos, para por fim adicionar esses gráficos a um novo dashboard.



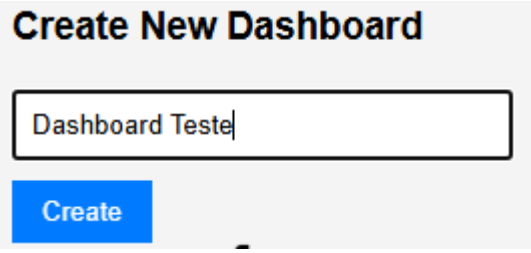
[**Figura 20**] Criação de sensor e tópicos com dados float e integer.



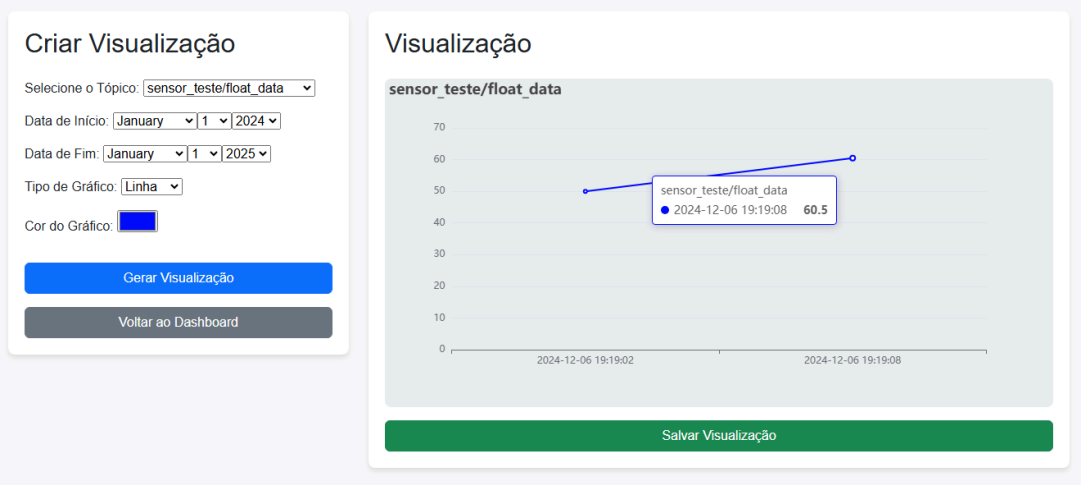
[**Figura 21**] Conexão com o broker MQTT iniciada.



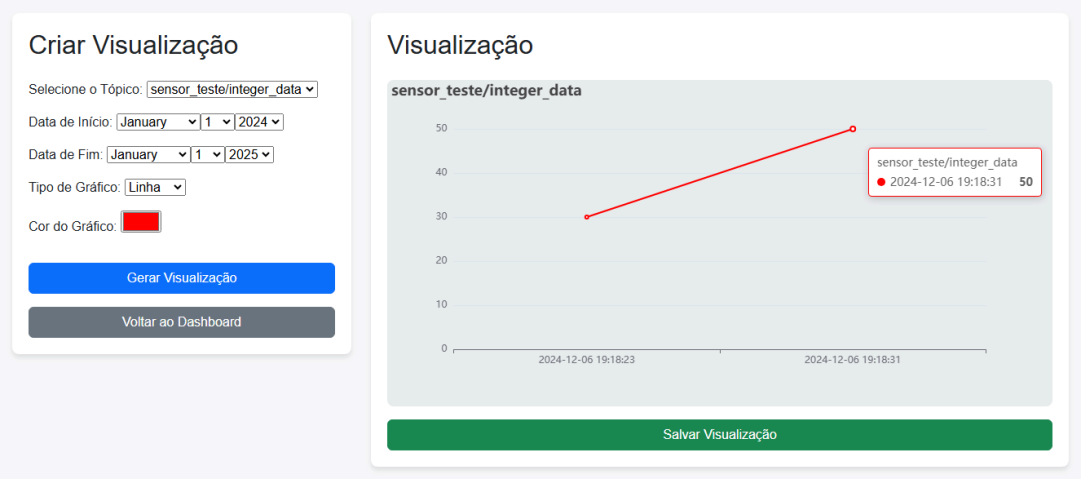
[**Figura 22**] Publicação de dados.



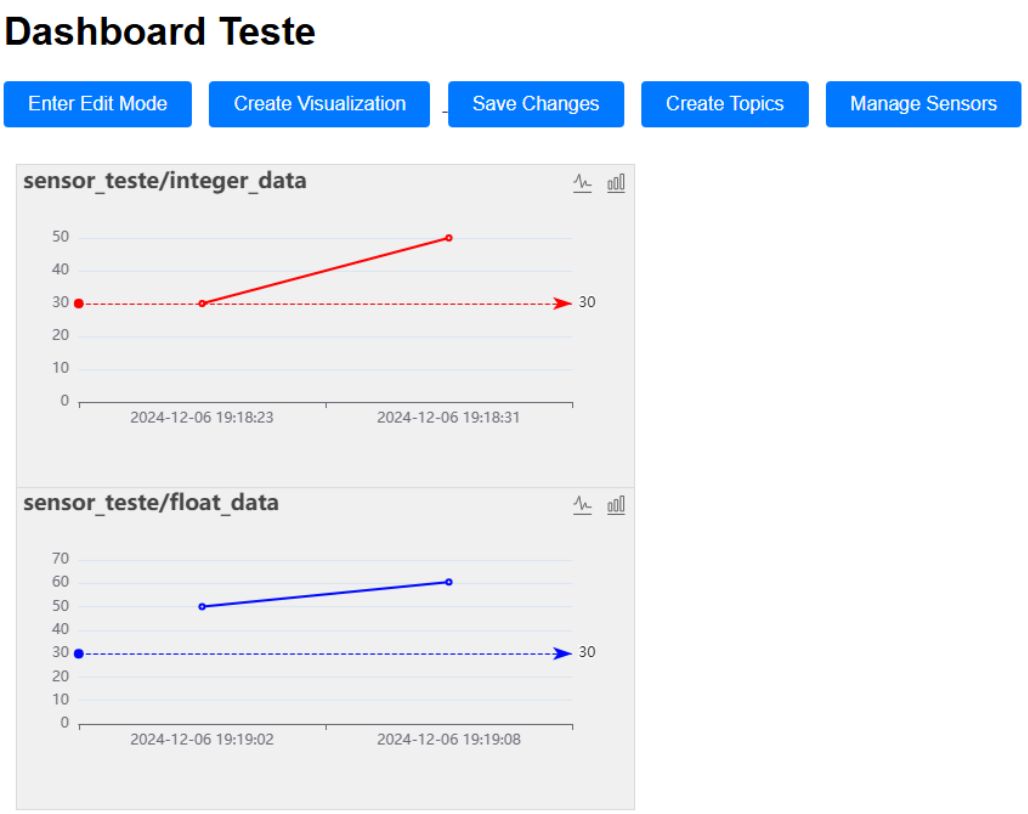
[**Figura 23**] Criação de Dashboard Teste.



[Figura 24] Criação de gráfico com tópico de dados float.



[**Figura 25**] Criação de gráficos com dados inteiros.



[**Figura 26**] Dashboards com paineis adicionados.

13. Interface de monitoramento dos sensores

O usuário Staff tem acesso a uma interface que permite o monitoramento dos sensores. Esse monitoramento se dá via um serviço systemd[5], que permite rodar o comando Django em background coletando os dados recebidos dos dispositivos por um broker MQTT. Para iniciar esse serviço foi necessário criar um arquivo de configuração:

```
[Unit]
Description=MQTT Monitoring Service for Sensor %i
After=network.target

[Service]
# Load environment variables from a file
EnvironmentFile=/etc/mqtt_sensor/%i.env

# Specify the Django settings module
Environment="DJANGO_SETTINGS_MODULE=setup.settings"

# Pass the sensor credentials from the environment variables
ExecStart=/usr/bin/python /home/dashboard/dashboard/manage.py mqtt_client
--sensor-name=%i --sensor-username=${SENSOR_USERNAME}
--sensor-password=${SENSOR_PASSWORD}

WorkingDirectory=/home/dashboard/dashboard
Restart=always
User=dashboard

[Install]
WantedBy=multi-user.target
```

Esse arquivo permite iniciar um serviço genérico para qualquer sensor, rodando o comando “manage.py mqtt_client”, e utilizando o nome do sensor e suas credenciais, que ficam em um arquivo de environment. Exemplo de arquivo de environment armazenado em **/etc/mqtt_sensor/teste2.env**.

```
SENSOR_USERNAME=teste
SENSOR_PASSWORD=teste123
```

Também foi criado um script bash, que permite acesso a todas as opções possíveis de um serviço systemd. Também foi atribuída permissão sudo para esse arquivo, para que seja possível rodar suas opções através da interface da aplicação, pela função subprocess do Django.

Para garantir acesso sudo ao script bash primeiro foi necessário na pasta /etc/sudoers.d/ criar o arquivo **00_sudoers_dashboard**. O conteúdo do arquivo deve ser:

```
dashboard ALL=NOPASSWD: /usr/local/bin/monitor_control
```

Também é necessário rodar o comando “chmod 700 /usr/local/bin/monitor_control”.

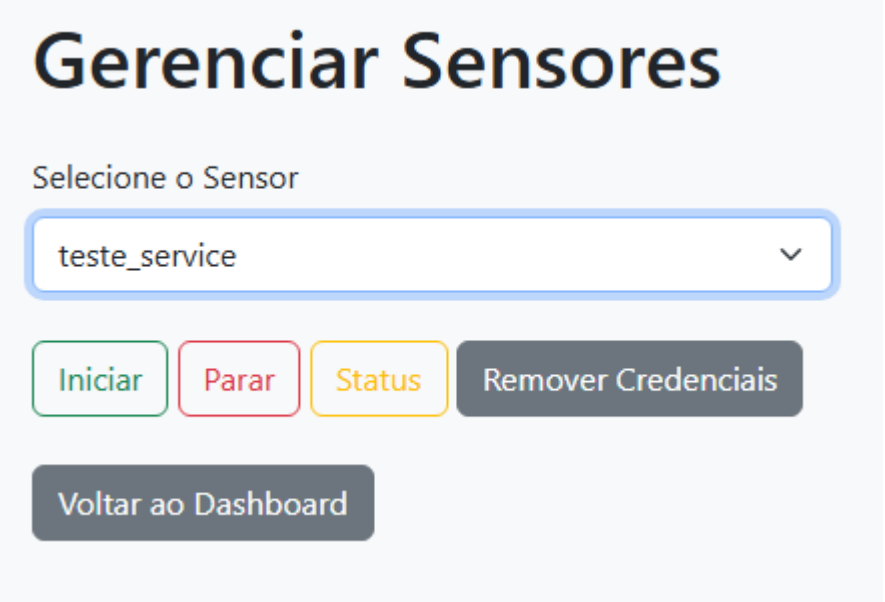
```
Usage: $0
<daemon-reload|start|enable|status|stop|create-credential|remove-credential>
<sensor-name> [username&&password]
```

```
# Process command
case "$1" in
    daemon-reload)
        sudo systemctl daemon-reload
        echo "Systemd daemon reloaded."
        ;;
    start)
        sudo systemctl start "$SERVICE_NAME"
        echo "Service started for sensor: $2"
        ;;
    enable)
        sudo systemctl enable "$SERVICE_NAME"
        echo "Service enabled for sensor: $2"
        ;;
    status)
        sudo systemctl status "$SERVICE_NAME"
        ;;
    stop)
        sudo systemctl stop "$SERVICE_NAME"
        echo "Service stopped for sensor: $2"
        ;;
    create-credential)
        create_credential "$2" "$3"
        ;;
    remove-credential)
        remove_credential "$2"
        ;;
```

Através da interface é verificado se o sensor já possui suas credenciais criadas no arquivo de enviroment. Caso esse arquivo de credenciais não tenha sido criado, é possível criá-lo, caso o arquivo já exista o usuário pode ter acesso

ao status do serviço, iniciá-lo ou habilitar a opção “enable” que faz com que o serviço sempre se inicie novamente caso o sistema de um reboot.

[**Figura 27**] Sensor com arquivo de credenciais não existente.



[**Figura 28**] Sensor com arquivo de credenciais existente.

14. Deploy

Para o deploy da aplicação Django utilizou-se Gunicorn[8] como servidor de aplicação, Nginx como proxy reverso, e Certbot para adicionar suporte a HTTPS com certificados SSL gratuitos.

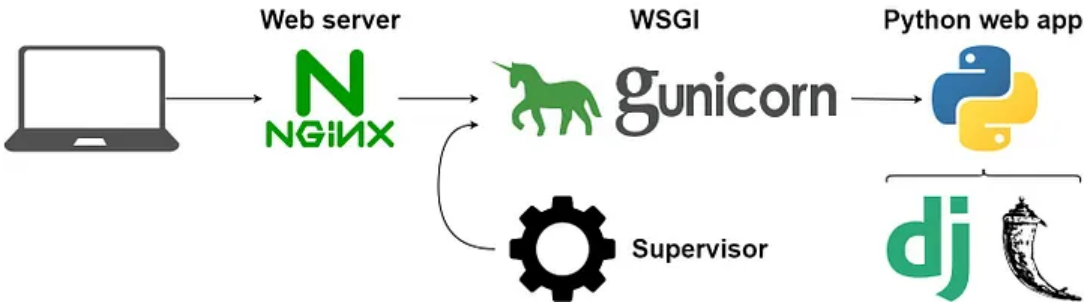
Antes do deploy algumas configurações do settings.py do Django precisam ser alteradas. Primeiramente a variável de Debug deve ser alterada para False. A Secret_key deve ser armazenada em uma variável de ambiente ao invés de ser hardcoded no arquivo. A variável Allowed_hosts deve ser alterada com os domínios necessários.Também é necessário configurar os arquivos estáticos e rodar o comando “python manage.py collectstatic”. As senhas do banco de dados também devem ser protegidas da mesma maneira que a Secret_key.

Após essas configurações do Django precisa-se configurar o Gunicorn. **Gunicorn** (abreviação de **Green Unicorn**) é um **servidor WSGI (Web Server Gateway Interface)** para aplicações web escritas em **Python**. Ele serve como uma ponte entre a aplicação Django (ou qualquer outra aplicação Python compatível com WSGI) e o servidor web, como o **Nginx**.

O Gunicorn provém melhor desempenho e segurança para a aplicação em ambiente de produção, fornecendo um servidor eficiente e robusto. Ele transforma as requisições HTTP recebidas do servidor Web (no caso o Nginx) em chamadas Python para a aplicação Django processar. Ele adota uma modelo de processo master com workers que são responsáveis por processar as requisições. O Gunicorn é fácil de configurar e pode ser ajustado para atender às necessidades do sistema. Ele é utilizado em conjunto com servidores Web como o Nginx. Para manter o Gunicorn rodando continuamente, vamos utilizar um serviço systemd da mesma forma que usamos para rodar os brokers MQTT dos sensores em background.

O Nginx é o servidor Web que foi utilizado. Ele é responsável por gerenciar as conexões HTTP e HTTPS, servir os arquivos estáticos como imagens e CSS, além de servir como proxy reverso para o Gunicorn.

Por último, utilizamos o **Certbot**, uma ferramenta automatizada de gerenciamento de certificados **SSL/TLS**, para gerar e configurar automaticamente certificados digitais. Esses certificados permitem que o servidor utilize uma conexão **HTTPS** segura, garantindo a criptografia dos dados transmitidos entre o cliente e o servidor, além de aumentar a confiança e a segurança do sistema.



[**Figura 29**] Diagrama do funcionamento do Gunicorn, Nginx e a aplicação Django.

15. Conclusão

Utilizar o framework Django permitiu um desenvolvimento mais rápido e seguro, por permitir uma conexão do backend com o frontend mais fácil e direta, além de ter muitos componentes integrados, como autenticação, ORM (fácil interação com os modelos e banco de dados), roteamento e gerência de formulários.

Usar o MQTT para a conexão com os dispositivos IoT também mostrou-se uma boa escolha, e junto com as funcionalidades do Django permitiu uma fácil conexão entre os dados e a inserção deles no banco, além de uma fácil visualização deles em gráficos customizáveis. A possibilidade de usar AJAX para atualizar os paineis e permitir o polling de dados sem a necessidade de atualizar a página também foi uma solução interessante para o problema da atualização em tempo real.

A integração do aplicativo com o sistema Linux também foi um ponto interessante do projeto, através do uso de serviços systemd e scripts bash, além do desenvolvimento total da aplicação na própria máquina virtual. Essa integração do Django com Linux é mais segura pois restringe as ações que podem ser executadas no sistema operacional. Apenas as tarefas pré-definidas pelo script bash, através das opções de execução disponíveis, podem ser executadas. Esse script é o único que pode ser executado com permissão de administração via sudo para ativar ou desativar serviços no systemd. Nenhuma outra tarefa administrativa pode ser acionada.

O uso das bibliotecas open-source para visualização de gráficos e personalização dos dashboards também ajudou na velocidade do desenvolvimento, além de possuírem funcionalidades muito úteis e versáteis.

Por fim, utilizar o Gunicorn para o deployment foi uma escolha confiável e eficiente. Ele se destacou pela facilidade de configuração e pela robustez em ambientes de produção, atuando como um intermediário entre o servidor e a aplicação Django. Aliado ao Nginx, que funciona como um proxy reverso, a solução proporcionou uma arquitetura otimizada e segura.

16. Trabalhos Futuros

A aplicação permite atualmente a visualização de dados de diversos dispositivos IoT em tempo real através de múltiplos dashboards e paineis com gráficos diferentes, além de um sistema de autenticação de usuários, e registro e monitoramento de sensores.

Trabalhos futuros podem envolver a colaboração de usuários em tempo real para construir os dashboards e as visualizações juntos, utilizando algoritmos de programação distribuída e concorrência.

Outro trabalho interessante é utilizar inteligência artificial e aprendizado de máquina para analisar os dados recebidos, gerando e entregando aos usuários informações interessantes.

Também podemos pensar em utilizar um framework de frontend como Angular, React ou Vue e o Django apenas como uma api REST somente para o backend. Poderíamos então explorar ideias interessantes como usar arquiteturas de infra, como Docker e Kubernetes para realizar a conexão do front com o backend.

Outra ideia interessante é adaptar a aplicação para interfaces mobile, dando mais liberdade ao usuário de utilizar o sistema e visualizar os dados.

Para adicionar mais segurança para a aplicação, a implantação de um sistema de autenticação em duas etapas Time-based One-Time Password (TOTP) com Google Authenticator ou Microsoft Authenticator para o login também é importante.

Atualmente os tópicos possuem limites superiores e inferiores visuais, porém o sistema de alerta da ultrapassagem do limiar ainda não foi implementado e será implementado futuramente.

O monitoramento de tópicos com dados do tipo JSON também será implementado futuramente.

17. Referências bibliográficas

[1] Django Software Foundation. **Django Documentation.** 2024. Disponível em: <https://docs.djangoproject.com/en/5.1/>

[2] The Apache Software Foundation. **The Apache Echarts.** 2024. Disponível em <https://echarts.apache.org/en/index.html>

[3] Gridstack Js Team. **Gridstack.js.** 2024. Disponível em: <https://gridstackjs.com/#>

[4] LIMA, Guilherme. **Django: templates e boas práticas**. 2024. Disponível em: <https://www.alura.com.br/curso-online-django-templates-boas-praticas>

[5] MOREL, Benjamin. **Creating a Linux Service with Systemd**. 2017. Disponível em: <https://medium.com/@benmorel/creating-a-linux-service-with-systemd-611b5c8b91d6>

[6] SCHWARZMULLER, Maximilian. **Python Django - The Pratical Guide.** 2024. Disponível em: <https://www.udemy.com/course/python-django-the-practical-guide/>

[7] Eclipse Foundation. **Paho MQTT.** 2024. Disponível em: <https://pypi.org/project/paho-mqtt/>

[8] ILARSLAN, Serdar. **What is Gunicorn**. 2022. Disponível em <https://medium.com/@serdarilarslan/what-is-gunicorn-5e674fff131b>

[9] SCHMEDTMANN, Jonas. **Build Responsive Real-World Websites with HTML and CSS.** 2024. Disponível em <https://www.udemy.com/course/design-and-develop-a-killer-website-with-html5-and-css3/?couponCode=BFCPSALE24>

[10] Wikipedia. **Grafana**. 2024. Disponível em <https://en.wikipedia.org/wiki/Grafana>

[11] Wikipedia. **ThingSpeak**. 2024 Disponível em <https://en.wikipedia.org/wiki/ThingSpeak>

[12] EQMX Team. **Introduction to MQTT Publish-Subscribe Pattern**. 2024. Disponível em <https://www.emqx.com/en/blog/mqtt-5-introduction-to-publish-subscribe-model>