

Engenharia Informática e Telecomunicações

Disciplina Sistemas Distribuídos



Engenharia Informática e Telecomunicações

Relatório de Sistemas Distribuídos – Trabalho Prático



Realizado por:

Francisco Fonseca Pv26039

Gonçalo Amorim Pv26041

Patrick Santos Pv26841

Márcio Silva Pv26047

Lamego, 2025

ÍNDICE

INTRODUÇÃO	5
ENUNCIADO	6
1- REQUISITOS E TECNOLOGIAS UTILIZADAS.....	7
1.1 REQUISITOS DO SISTEMA.....	7
1.2 FERRAMENTAS E TECNOLOGIAS	8
1.2.1 DESENVOLVIMENTO	8
1.2.2 DEPLOYMENT.....	10
1.2.3 OUTRAS FERRAMENTAS RELEVANTES.....	11
2- ARQUITETURA DO SISTEMA.....	12
2.1 DESCRIÇÃO DA ARQUITETURA.....	12
2.1.1 SEPARAÇÃO CLIENTE SERVIDOR	12
2.1.2 COMUNICAÇÃO ENTRE OS COMPONENTES	12
2.2 DIAGRAMA DA ARQUITETURA.....	13
3- DESENVOLVIMENTO	14
3.1 CONFIGURAÇÃO DO AMBIENTE.....	14
3.2 IMPLEMENTAÇÃO	16
3.2.1 VIEWS	16
3.3 APIs REST	28
3.3.1 API de Sensores	28
3.3.2 API de Controlo do Estendal.....	30
3.3.3 API de Emparelhamento do Estendal.....	33
3.4 DESAFIOS E SOLUÇÕES.....	36
4- TESTES E RESULTADOS	38
4.1 DESCRIÇÃO DOS TESTES.....	38
4.2 RESULTADOS OBTIDOS.....	40
CONCLUSÃO.....	43
BIBLIOGRAFIA	44

INDICE IMAGENS

Figura 1 - Enunciado	6
Figura 2 - Docker Containers	13
Figura 3 - Docker Compose - Django	14
Figura 4 - Docker Compose - DB	15
Figura 5 - Imports.....	16
Figura 6 - Dashboard.....	17
Figura 7 - Login.....	18
Figura 8 - Logout.....	19
Figura 9 - Historico	20
Figura 10 – Definições/1	22
Figura 11 - Definições/2.....	23
Figura 12 - Definições/3.....	24
Figura 13 - Remover Estendal.....	25
Figura 14 – Registo/1	26
Figura 15 - Registo/2.....	26
Figura 16 - API Sensors	28
Figura 17 - API Control/1	30
Figura 18 - API Control/2	31
Figura 19 - API Pair/1	33
Figura 20 - API Pair/2	34
Figura 21 - Dashboard.....	40
Figura 22 - Histórico	40
Figura 23 - Definições.....	41
Figura 24 - Teste de API Sensors.....	41
Figura 25 - Teste de API Pair.....	42
Figura 26 - Teste de API Control	42

INTRODUÇÃO

O presente relatório tem como objetivo descrever o desenvolvimento de uma aplicação web destinada à monitorização e controlo de um estendal inteligente, concebida segundo os princípios fundamentais das arquiteturas distribuídas.

Ao longo do documento são abordados, de forma estruturada, os principais aspetos inerentes ao projeto desenvolvido. Inicialmente, é apresentado o enunciado do trabalho, seguido da identificação dos requisitos funcionais e não funcionais do sistema, bem como das tecnologias e ferramentas adotadas durante o processo de desenvolvimento e deployment. Nesta fase, é também justificada a escolha das soluções tecnológicas utilizadas, com destaque para o recurso ao framework Django e à utilização de containers Docker.

Posteriormente, é descrita a arquitetura do sistema, evidenciando a separação cliente–servidor, a comunicação entre os diferentes componentes e a organização modular baseada em containers. Esta secção inclui ainda a análise do modelo de comunicação adotado, nomeadamente através de APIs REST e da integração com serviços externos.

O relatório prossegue com a descrição do processo de desenvolvimento, contemplando a configuração do ambiente, a implementação das principais funcionalidades da aplicação, a gestão de utilizadores, a apresentação dos dados recolhidos pelos sensores e o controlo do estendal inteligente. É dada especial atenção à implementação das APIs REST, responsáveis pela comunicação entre o backend, os dispositivos físicos e outros serviços.

Adicionalmente, são apresentados os desafios identificados durante o desenvolvimento do projeto, bem como as soluções adotadas para os ultrapassar, permitindo evidenciar o processo de tomada de decisão e a aplicação prática dos conceitos teóricos da disciplina.

Por fim, o relatório aborda a fase de testes e resultados, onde são descritos os testes unitários e de integração realizados, assim como os resultados obtidos, permitindo validar o correto funcionamento do sistema.

ENUNCIADO

Sistemas Distribuídos Trabalho Prático

Data-limite de entrega: 16 de dezembro de 2025

Apresentação: 17 de dezembro de 2025

1. Pretende-se que os alunos criem uma aplicação.
2. Devem utilizar para a comunicação entre cliente / servidor pelo menos uma das tecnologias abordadas na UC.
3. O trabalho pode ser realizado em grupo, até um máximo de 4 elementos.
4. Qualquer semelhança entre programas de qualquer grupo implicará a anulação dos mesmos.
5. Avaliação do trabalho :
 - a. Desenvolvimento – 8 pontos.
 - b. Desenvolvimento gráfico (interface gráfico) – 4 pontos.
 - c. Relatório – 5 pontos.
 - d. Apresentação – 3 pontos.

Figura 1 - Enunciado

1- REQUISITOS E TECNOLOGIAS UTILIZADAS

1.1 REQUISITOS DO SISTEMA

A aplicação desenvolvida consiste num website destinado à monitorização e controlo de um estendal inteligente, cujo desenvolvimento decorre no âmbito da unidade curricular de Sistemas de Telecomunicações. O sistema foi implementado utilizando o framework Django, seguindo uma arquitetura distribuída baseada em containers Docker.

A aplicação é composta por dois containers principais: um responsável pelo backend da aplicação (Clothesline_Django) e outro destinado à base de dados (Clothesline_BD). Esta abordagem permite uma melhor separação de responsabilidades, facilitando o deployment e a escalabilidade do sistema.

Do ponto de vista do utilizador, a aplicação disponibiliza um sistema de autenticação, permitindo o registo e o login de utilizadores. Após autenticação, o utilizador tem acesso a três áreas principais da aplicação: Dashboard, Histórico e Definições.

A Dashboard apresenta, em tempo quase real, os valores recolhidos pelos sensores do estendal inteligente, permitindo igualmente o controlo manual do sistema, nomeadamente a abertura e o fecho do estendal. Adicionalmente, integra um widget informativo que recorre à API do IPMA, possibilitando a consulta de dados meteorológicos de diferentes regiões.

A página de Histórico disponibiliza gráficos representativos da temperatura e da humidade/vento registados nas últimas 24 horas, bem como uma secção dedicada ao registo de eventos recentes, tais como a deteção de chuva ou ações manuais realizadas pelo utilizador.

Por fim, a área de Definições permite ao utilizador consultar informações pessoais, como o nome de utilizador e o endereço de correio eletrónico, bem como configurar determinados parâmetros associados ao funcionamento do estendal inteligente.

De forma resumida, os principais requisitos funcionais do sistema são:

- ✓ Autenticação e registo de utilizadores;
- ✓ Dashboard com visualização de dados em tempo quase real;
- ✓ Controlo manual do estendal;
- ✓ Integração com uma API externa (IPMA);
- ✓ Visualização de histórico através de gráficos;
- ✓ Registo e consulta de eventos;
- ✓ Página de definições do utilizador e do sistema.

A comunicação entre o backend desenvolvido em Django e a base de dados é realizada através do ORM (Object-Relational Mapping) disponibilizado pelo framework, permitindo a persistência e recuperação de dados de forma abstrata e segura.

Internamente, a aplicação segue o padrão arquitetural MVT (Model–View–Template), promovendo a separação entre a lógica de negócio, a apresentação e o acesso aos dados. Adicionalmente, foram desenvolvidas APIs REST, que possibilitam a troca de dados entre o backend e outros componentes do sistema, assegurando uma comunicação eficiente e compatível com um ambiente distribuído.

1.2 FERRAMENTAS E TECNOLOGIAS

1.2.1 DESENVOLVIMENTO

O desenvolvimento da aplicação foi realizado maioritariamente em Python, utilizando o framework Django para a implementação do backend. A escolha desta linguagem e framework deveu-se ao maior domínio da equipa sobre estas tecnologias, bem como à sua robustez, maturidade e facilidade de integração com bases de dados relacionais e APIs externas, características fundamentais no contexto de sistemas distribuídos.

Para o desenvolvimento da interface web foram utilizadas linguagens e tecnologias padrão da Web, nomeadamente:

- HTML (templates Django), para a estruturação das páginas da aplicação;
- CSS, recorrendo ao framework Tailwind CSS, para a estilização da interface, permitindo um design moderno e responsivo;
- JavaScript, para implementar interações dinâmicas entre o utilizador e a aplicação web.

Frameworks e Bibliotecas Relevantes

O projeto recorre a um conjunto de frameworks e bibliotecas que suportam as diferentes funcionalidades da aplicação, destacando-se as seguintes:

✓ Django	(versão	5.2.8)
	Framework web principal utilizado no desenvolvimento do backend, responsável pela gestão de rotas, autenticação de utilizadores, implementação da lógica de negócio e interação com a base de dados através do ORM.	

- ✓ **Django REST Framework** (versão 3.16.1)
Utilizado para a criação de APIs REST, permitindo a comunicação e troca de dados entre o frontend, o backend e outros componentes do sistema distribuído.

- ✓ **django-rest-passwordreset** (versão 1.5.0)
Biblioteca utilizada para implementar funcionalidades relacionadas com a recuperação e redefinição de palavras-passe dos utilizadores.

- ✓ **django-tailwind (versão 4.4.1) e pytailwindcss (versão 0.3.0)**
Ferramentas responsáveis pela integração do framework Tailwind CSS no projeto Django, possibilitando uma estilização eficiente, consistente e responsiva da interface gráfica.

- ✓ **django-browser-reload** (versão 1.21.0)
Ferramenta de apoio ao desenvolvimento, utilizada para facilitar a atualização automática das páginas durante a fase de implementação, aumentando a produtividade da equipa.

- ✓ **requests** (versão 2.32.5)
Biblioteca utilizada para o consumo de APIs externas, nomeadamente a API do IPMA, permitindo a obtenção de dados meteorológicos integrados na aplicação.

- ✓ **psycopg2-binary** (versão 2.9.11)
Driver responsável pela comunicação entre a aplicação Django e a base de dados PostgreSQL, assegurando uma ligação eficiente e fiável.

Adicionalmente, foram utilizadas outras bibliotecas auxiliares, tais como python-dateutil, PyYAML, sqlparse e python-slugify, que dão suporte ao processamento de datas, gestão de configurações, manipulação de dados e ao funcionamento interno da aplicação.

1.2.2 DEPLOYMENT

Docker para Deployment

O deployment da aplicação é realizado recorrendo à plataforma Docker, permitindo a execução do sistema em ambientes isolados, consistentes e reprodutíveis, independentemente do sistema operativo subjacente. Esta abordagem facilita não só o processo de instalação e execução da aplicação, como também a sua manutenção e escalabilidade, aspectos particularmente relevantes em sistemas distribuídos.

Foram definidos dois containers principais, cada um com responsabilidades bem delimitadas:

- ✓ **Clothesline_Django:** container responsável pela execução do backend da aplicação web, onde corre o servidor Django e onde se encontra implementada a lógica de negócio, bem como as APIs REST disponibilizadas pelo sistema.
- ✓ **Clothesline_BD:** container dedicado à base de dados, assegurando a persistência da informação e garantindo a separação entre a camada de aplicação e a camada de dados.

A comunicação entre os containers é realizada através da rede interna definida pelo Docker, permitindo que o backend aceda à base de dados de forma transparente e segura. Para que a aplicação web seja corretamente disponibilizada, é necessário que ambos os containers estejam em execução, uma vez que o funcionamento do sistema depende da interação entre a lógica da aplicação e a persistência dos dados.

Esta arquitetura baseada em containers contribui para uma maior modularidade do sistema, simplificando o processo de deployment e reforçando os princípios de separação de responsabilidades e escalabilidade característicos de aplicações distribuídas.

1.2.3 OUTRAS FERRAMENTAS RELEVANTES

Para além das tecnologias utilizadas no desenvolvimento e deployment da aplicação, foram igualmente adotadas diversas ferramentas complementares que suportam o funcionamento, a integração e a validação do sistema distribuído.

- ✓ **PostgreSQL**

Sistema de gestão de base de dados relacional utilizado para o armazenamento persistente dos dados da aplicação, incluindo informações dos utilizadores, valores recolhidos pelos sensores, registo de eventos e histórico de funcionamento do estendal inteligente. A sua escolha deveu-se à fiabilidade, desempenho e boa integração com o framework Django.

- ✓ **API do IPMA**

Serviço externo utilizado para a obtenção de dados meteorológicos, permitindo à aplicação apresentar informações climáticas atualizadas de diferentes regiões. A integração desta API enriquece a funcionalidade do sistema, fornecendo contexto adicional para a monitorização e controlo do estendal.

- ✓ **Git**

Sistema de controlo de versões utilizado ao longo de todo o desenvolvimento do projeto, possibilitando a gestão do código-fonte, o trabalho colaborativo entre os elementos do grupo e o histórico de alterações efetuadas.

- ✓ **Postman**

Ferramenta utilizada para testar e validar os endpoints das APIs REST desenvolvidas, garantindo o correto funcionamento da comunicação entre os diferentes componentes do sistema e auxiliando na deteção de erros durante a fase de desenvolvimento.

2- ARQUITETURA DO SISTEMA

2.1 DESCRIÇÃO DA ARQUITETURA

2.1.1 SEPARAÇÃO CLIENTE SERVIDOR

Neste projeto foi adotada uma arquitetura cliente–servidor, na qual as responsabilidades do sistema se encontram claramente separadas entre o cliente e o servidor, promovendo modularidade, escalabilidade e facilidade de manutenção.

O cliente corresponde à componente web da aplicação, através da qual o utilizador acede ao sistema utilizando um navegador. Esta componente é responsável por apresentar a interface gráfica da aplicação, sob a forma de páginas web, enviar pedidos ao servidor, tais como autenticação, consulta de dados dos sensores ou comandos de controlo do estendal e receber e apresentar ao utilizador as respostas devolvidas pelo servidor.

O servidor corresponde ao backend da aplicação, desenvolvido em Django e executado em ambiente Docker, estando organizado em dois containers distintos: um dedicado à lógica da aplicação e outro à base de dados. Esta componente é responsável por processar os pedidos recebidos do cliente, executar a lógica de negócio da aplicação, gerir os mecanismos de autenticação e autorização dos utilizadores, comunicar com a base de dados para a persistência e recuperação de informação e fornecer dados ao cliente sob a forma de páginas web renderizadas, quer através de respostas provenientes das APIs REST disponibilizadas pelo sistema.

Esta separação entre cliente e servidor permite uma clara divisão de responsabilidades, facilitando a evolução do sistema e reforçando os princípios fundamentais das arquiteturas distribuídas.

2.1.2 COMUNICAÇÃO ENTRE OS COMPONENTES

A comunicação entre o cliente e o servidor é realizada através do protocolo HTTP, recorrendo a métodos de pedido adequados ao modelo cliente–servidor. No contexto deste projeto, foi utilizado o método POST para o envio de dados do cliente para o servidor, nomeadamente em operações relacionadas com autenticação, controlo do estendal e obtenção de informação proveniente dos sensores.

Os dados trocados entre o cliente e o servidor são estruturados no formato JSON, permitindo uma representação leve e independente de plataforma. Estes dados são processados através de APIs REST desenvolvidas no backend, responsáveis por disponibilizar funcionalidades como

a recolha de dados dos sensores, o controlo do estendal e o emparelhamento (pairing) de dispositivos. Para além das APIs internas, a aplicação integra igualmente uma API externa, disponibilizada pelo IPMA, utilizada para a obtenção de dados meteorológicos.

A comunicação entre o servidor Django e a base de dados é efetuada através do ORM (Object-Relational Mapping) do Django, que traduz automaticamente as operações definidas em Python em comandos SQL. Estes comandos são executados sobre uma base de dados PostgreSQL, garantindo uma interação eficiente, segura e abstrata com a camada de persistência de dados.

Esta abordagem de comunicação assegura uma clara separação entre as diferentes camadas do sistema, promovendo a interoperabilidade e alinhando-se com os princípios de desenvolvimento de sistemas distribuídos.

2.2 DIAGRAMA DA ARQUITETURA

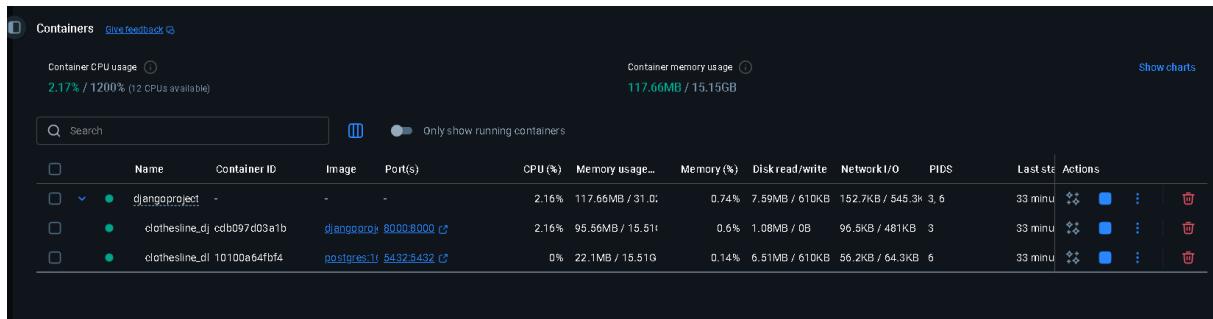


Figura 2 - Docker Containers

O diagrama apresentado ilustra a arquitetura da aplicação em execução num ambiente Docker, evidenciando a utilização de múltiplos containers para suportar o funcionamento do sistema distribuído. Em particular, é possível observar a separação entre o container responsável pelo backend da aplicação Django e o container dedicado à base de dados PostgreSQL, ambos em execução simultânea.

Esta organização demonstra a adoção de uma arquitetura modular baseada em containers, na qual cada componente possui responsabilidades bem definidas e comunica através da rede interna do Docker. A execução isolada dos serviços contribui para uma maior fiabilidade, facilidade de deployment e escalabilidade do sistema, alinhando-se com os princípios fundamentais das arquiteturas cliente–servidor e de sistemas distribuídos.

3- DESENVOLVIMENTO

3.1 CONFIGURAÇÃO DO AMBIENTE

Para a configuração do ambiente de execução da aplicação foi utilizado o Docker Compose, em conjunto com um Dockerfile, permitindo automatizar o processo de criação e orquestração dos containers necessários ao funcionamento do sistema. Esta abordagem facilita a reprodução do ambiente de desenvolvimento e assegura consistência entre diferentes execuções.

O ficheiro Docker Compose foi configurado para criar e gerir dois containers principais: clothesline_django e clothesline_db, correspondentes, respetivamente, ao backend da aplicação e à base de dados.

clothesline_django:

O container clothesline_django possui diversos parâmetros essenciais ao seu funcionamento. O parâmetro command é utilizado para executar o servidor de desenvolvimento do Django através do comando python manage.py runserver 0.0.0.0:8000, permitindo que a aplicação esteja acessível externamente ao container. O parâmetro ports é utilizado para mapear a porta interna do container para a porta 8000 do sistema anfitrião, possibilitando o acesso à aplicação via navegador. Adicionalmente, é definido um parâmetro volumes, responsável pela criação de um volume denominado app, que permite a partilha e persistência dos ficheiros da aplicação. Por fim, o parâmetro environment é utilizado para definir as variáveis de ambiente necessárias, incluindo as dependências e credenciais de ligação ao container da base de dados, garantindo a correta comunicação entre os serviços.

```
services:
  django:
    build: .
    container_name: clothesline_django
    command: python manage.py runserver 0.0.0.0:8000
    ports:
      - "8000:8000"
    volumes:
      - .:/app
    environment:
      - DB_HOST=db
      - DB_NAME=clothesline
      - DB_USER=postgres
      - DB_PASSWORD=postgres
      - DEBUG=1
    depends_on:
      - db
```

Figura 3 - Docker Compose - Django

clothesline_db:

O container clothesline_db, responsável pela base de dados, apresenta uma configuração mais simples, uma vez que a sua função se restringe à persistência e gestão dos dados da aplicação. Este container é identificado pelo seu respetivo nome e encontra-se configurado com a política de reinício automático, através do parâmetro restart, garantindo que o serviço é reiniciado em caso de falha.

O parâmetro ports é utilizado para mapear a porta padrão da base de dados PostgreSQL (5432), permitindo a comunicação com o container responsável pelo backend da aplicação. A ligação entre o container da base de dados e o container clothesline_django é assegurada através da rede definida no Docker Compose, possibilitando uma comunicação direta e eficiente entre ambos.

Adicionalmente, é criado um volume dedicado, denominado postgres_data, que permite a persistência dos dados da base de dados, assegurando que a informação armazenada se mantém mesmo após a paragem ou reinício dos containers.

Esta configuração contribui para a fiabilidade e integridade dos dados do sistema, reforçando a separação de responsabilidades entre a camada de aplicação e a camada de persistência.

```
db:
  image: postgres:16
  container_name: clothesline_db
  restart: always
  ports:
    - "5432:5432"
  environment:
    POSTGRES_DB: clothesline
    POSTGRES_USER: postgres
    POSTGRES_PASSWORD: postgres
  volumes:
    - postgres_data:/var/lib/postgresql/data

volumes:
  postgres_data:
```

Figura 4 - Docker Compose - DB

3.2 IMPLEMENTAÇÃO

3.2.1 VIEWS

```
from django.contrib.auth.models import User
from django.http import HttpResponseRedirect
from django.shortcuts import render, redirect
from django.contrib.auth import authenticate, login as logar, logout as sair
from django.contrib.auth.decorators import login_required
from .models import DryingRack, Sensor, Alert
import re
```

Figura 5 - Imports

O excerto de código apresentado corresponde aos imports utilizados nas views da aplicação Django, sendo responsáveis por disponibilizar as funcionalidades necessárias ao processamento dos pedidos do cliente.

São importados módulos do Django relacionados com a gestão de utilizadores e autenticação, nomeadamente o modelo User, bem como funções para autenticação, login e logout.

Adicionalmente, são utilizadas classes e métodos para a renderização de templates, redirecionamento de pedidos HTTP e controlo de acesso às views, através do decorador login_required, garantindo que determinadas funcionalidades apenas estão acessíveis a utilizadores autenticados.

Foram igualmente importados os modelos definidos na aplicação (DryingRack, Sensor e Alert), que representam as entidades principais do sistema, permitindo a interação com a base de dados através do ORM do Django.

Por fim, é utilizada a biblioteca re para suporte a operações de processamento de texto recorrendo a expressões regulares.

Este conjunto de imports assegura a integração entre a camada de apresentação, a lógica de negócio e a persistência de dados, contribuindo para a correta implementação das funcionalidades da aplicação.

```

@login_required(login_url="/login/")
def dashboard(request):
    rack = (
        DryingRack.objects
        .filter(user=request.user, active=True)
        .order_by("installation_date")
        .first()
    )

    sensor = None
    if rack:
        # Última leitura de sensores do estendal
        sensor = (
            Sensor.objects
            .filter(drying_rack=rack)
            .order_by("-datetime")
            .first()
        )

    # Último alerta de qualquer estendal do utilizador
    last_alert = (
        Alert.objects
        .filter(drying_rack__user=request.user)
        .order_by("-datetime")
        .first()
    )

    context = {
        "rack": rack,
        "sensor": sensor,
        "last_alert": last_alert,
    }
    return render(request, template_name="estendal/dashboard.html", context)

```

Figura 6 - Dashboard

O excerto de código apresentado corresponde à view responsável pela Dashboard da aplicação. Esta view encontra-se protegida pelo decorador `login_required`, garantindo que apenas utilizadores autenticados podem aceder a esta funcionalidade.

Numa primeira fase, é obtido o estendal associado ao utilizador autenticado, filtrando apenas os estendais ativos e selecionando o mais recente com base na data de instalação. Caso exista um estendal associado ao utilizador, é efetuada a recolha da última leitura dos sensores correspondente a esse estendal, permitindo apresentar informação atualizada na interface.

Adicionalmente, é obtido o último alerta registado associado a qualquer estendal do utilizador, possibilitando a notificação de eventos relevantes, tais como alterações de estado ou condições ambientais adversas.

Por fim, os dados recolhidos, nomeadamente o estendal, os valores dos sensores e o último alerta, são agregados num objeto de contexto e enviados para o template HTML da Dashboard.

Este processo permite a apresentação dinâmica da informação ao utilizador, assegurando a separação entre a lógica de negócio e a camada de apresentação, de acordo com o padrão MVT adotado pelo framework Django.

```
def login(request):
    if request.method == "GET":
        return render(request, template_name: "sistema_login/login.html")

    username = request.POST.get(key: "username", default: "")
    password = request.POST.get(key: "password", default: "")

    user = authenticate(request, username=username, password=password)

    if user is None:
        return render(request, template_name: "sistema_login/login.html", context: {
            "error": "Username ou password incorretos.",
            "username": username
        })

    logar(request, user)
    return redirect("dashboard")
```

Figura 7 - Login

O excerto de código apresentado corresponde à view responsável pelo processo de autenticação (Login) dos utilizadores na aplicação.

Inicialmente, quando é efetuado um pedido do tipo GET, a view limita-se a renderizar a página de login, apresentando o formulário de autenticação ao utilizador. Quando o pedido é do tipo POST, são recolhidas as credenciais introduzidas pelo utilizador, nomeadamente o nome de utilizador e a palavra-passe.

De seguida, estas credenciais são validadas através da função authenticate, disponibilizada pelo Django. Caso o utilizador não exista ou as credenciais estejam incorretas, a página de login é novamente apresentada, sendo exibida uma mensagem de erro informativa. Caso a autenticação seja bem-sucedida, o utilizador é autenticado através da função login e redirecionado para a Dashboard da aplicação.

Este mecanismo assegura um processo de autenticação seguro e consistente, utilizando as funcionalidades nativas do Django para a gestão de utilizadores e controlo de acesso.

```
def logout(request):
    sair(request)
    return redirect("home")
```

Figura 8 - Logout

A aplicação dispõe igualmente de uma view dedicada ao processo de logout dos utilizadores. Nesta view é utilizada a funcionalidade nativa de logout disponibilizada pelo framework Django, a qual foi importada com o alias sair.

Quando esta view é invocada, a sessão do utilizador autenticado é terminada de forma segura, removendo as credenciais associadas à sessão ativa. Após a execução do logout, o utilizador é redirecionado para a página inicial da aplicação.

Este mecanismo garante um controlo adequado das sessões de utilizador, contribuindo para a segurança e correta gestão do acesso à aplicação.

```

@login_required(login_url="/login/")
def historico(request):
    rack = (
        DryingRack.objects
        .filter(user=request.user, active=True)
        .order_by("installation_date")
        .first()
    )

    sensors = []
    alerts = []

    if rack:
        last_24h = timezone.now() - timedelta(hours=24)

        sensors = [
            {
                "datetime": s.datetime.isoformat(),
                "temperature": s.temperature,
                "humidity": s.humidity,
            }
            for s in Sensor.objects.filter(
                drying_rack=rack,
                datetime__gte=last_24h
            ).order_by("datetime")
        ]

        alerts = (
            Alert.objects
            .filter(drying_rack=rack)
            .order_by("-datetime")[:20]
        )

    context = {
        "sensors": sensors, # ✓ LISTA JSON
        "alerts": alerts,
        "rack": rack,
    }

    return render(request, "estendal/historico.html", context)

```

Figura 9 - Historico

O excerto de código acima corresponde à view responsável pela página de Histórico da aplicação, a qual se encontra protegida pelo decorador login_required, garantindo o acesso exclusivo a utilizadores autenticados.

Numa fase inicial, é identificado o estendal ativo associado ao utilizador, seguindo o mesmo critério utilizado noutras views da aplicação. De seguida, são inicializadas listas destinadas ao armazenamento dos valores dos sensores e dos alertas, que posteriormente serão utilizadas para a apresentação de gráficos e tabelas informativas.

Caso exista um estendal associado ao utilizador, a view procede à recolha das leituras dos sensores registadas nas últimas 24 horas, organizando os dados relevantes, tais como data/hora, temperatura e humidade, num formato estruturado adequado à sua representação gráfica. Paralelamente, são obtidos os alertas mais recentes associados ao estendal, ordenados por data e limitados a um número definido de registo, de forma a facilitar a sua visualização.

Por fim, os dados relativos ao estendal, às leituras dos sensores e aos alertas são agregados num objeto de contexto, que é enviado para o template HTML correspondente à página de Histórico. Este processo permite a apresentação dinâmica da informação ao utilizador, mantendo a separação entre a lógica de negócio e a camada de apresentação, em conformidade com o padrão MVT adotado na aplicação.

```

def definicoes(request):
    user = request.user

    rack = (
        DryingRack.objects
        .filter(user=user, active=True)
        .first()
    )

    modoAutomatico = [
        {
            "label": "Ativar Modo Automático",
            "desc": "Gestão automática baseada nas condições meteorológicas",
            "checked": True,
        },
        {
            "label": "Fechar com Chuva",
            "desc": "Fecha automaticamente quando a chuva é detetada",
            "checked": True,
        },
        {
            "label": "Fechar com Vento Forte",
            "desc": "Fecha quando o vento excede 30 km/h",
            "checked": True,
        },
    ]

    sensors = []
    errors = {}

    if rack:
        sensors = (
            Sensor.objects
            .filter(drying_rack=rack)
            .order_by('-datetime')[:100]
        )

```

Figura 10 – Definições/1

```

if request.method == "POST":
    nome = request.POST.get("nome")
    email = request.POST.get("email")
    nova_password = request.POST.get("nova_password")

    user.username = nome
    user.email = email

    password_alterada = False

    if nova_password:
        user.set_password(nova_password)
        password_alterada = True

    pwd_errors = []
    if nova_password:
        if len(nova_password) < 8:
            pwd_errors.append("Pelo menos 8 caracteres.")
        if not re.search(pattern: r"[A-Z]", nova_password):
            pwd_errors.append("Pelo menos 1 letra maiúscula.")
        if not re.search(pattern: r"[a-z]", nova_password):
            pwd_errors.append("Pelo menos 1 letra minúscula.")
        if not re.search(pattern: r"\d", nova_password):
            pwd_errors.append("Pelo menos 1 número.")

    if pwd_errors:
        errors["password"] = " ".join(pwd_errors)

    if errors:
        context = {
            "errors": errors,
            "modoAutomatico": modoAutomatico,
            "rack": rack,
            "sensors": sensors,
        }
        return render(request, template_name: "estendal/definicoes.html", context)

```

Figura 11 - Definições/2

```

    user.save()

    if password_alterada:
        return redirect("/logout/")

    messages.success(request, "Definições atualizadas com sucesso.")
    return redirect("definicoes")

    return render(request, template_name="estendal/definicoes.html", context={
        "errors": errors,
        "modoAutomatico": modoAutomatico,
        "rack": rack,
        "sensors": sensors,
    })
}

```

Figura 12 - Definições/3

O conjunto de excertos de código apresentados corresponde à view responsável pela página de Definições da aplicação. Esta view agrega tanto a lógica de apresentação da página como a lógica de atualização dos dados do utilizador, consoante o tipo de pedido HTTP efetuado.

Numa fase inicial, são obtidos o utilizador autenticado e o estendal ativo associado, sendo igualmente preparadas as opções de configuração do modo automático do estendal. Adicionalmente, são recolhidas as leituras mais recentes dos sensores, de forma a disponibilizar informação de apoio à configuração do sistema.

Quando é efetuado um pedido do tipo POST, a view trata a submissão do formulário de Definições, permitindo a atualização dos dados pessoais do utilizador, nomeadamente o nome de utilizador e o endereço de correio eletrónico. É igualmente suportada a alteração da palavra-passe, sendo aplicadas validações de segurança, como comprimento mínimo e complexidade (presença de letras maiúsculas, minúsculas e números).

Caso sejam detetados erros de validação, estes são comunicados ao utilizador e a página é novamente apresentada com as respetivas mensagens de erro. Quando a atualização é concluída com sucesso, os dados são guardados e o utilizador é notificado. No caso de alteração da palavra-passe, a sessão é terminada, obrigando a uma nova autenticação por motivos de segurança.

```
@login_required(login_url="/login/") new *
@require_POST
def remover_estendal(request):
    rack = (
        DryingRack.objects
        .filter(user=request.user, active=True)
        .first()
    )

    if rack:
        rack.active = False
        rack.save()

    return redirect("definicoes")
```

Figura 13 - Remover Estendal

O excerto de código apresentado corresponde à view responsável pela remoção (desativação) do estendal associado ao utilizador. Esta view encontra-se protegida pelos mecanismos de autenticação do Django, garantindo que apenas utilizadores autenticados podem executar esta ação, e aceita exclusivamente pedidos do tipo POST, prevenindo remoções acidentais através de acessos diretos por URL.

A view identifica o estendal ativo associado ao utilizador autenticado e, caso este exista, procede à sua desativação lógica, alterando o estado do atributo `active` para falso. Esta abordagem permite preservar os dados históricos associados ao estendal, evitando a eliminação física dos registo na base de dados.

Após a atualização do estado do estendal, o utilizador é redirecionado para a página de Definições. Este mecanismo assegura uma gestão controlada dos dispositivos associados a cada utilizador, reforçando a segurança e a integridade dos dados no sistema.

```

def register(request):
    if request.method == "GET":
        return render(request, template_name="sistema_login/registo.html")

    # POST
    username = request.POST.get(key="username", default "").strip()
    email = request.POST.get(key="email", default "").strip()
    password1 = request.POST.get(key="password1", default "")
    password2 = request.POST.get(key="password2", default "")

    errors = {}
    form_data = {"username": username, "email": email}

    # 1) Campos vazios
    if not username or not email or not password1 or not password2:
        errors["general"] = "Preencha todos os campos."

    # 2) Username já existe
    if username and User.objects.filter(username=username).exists():
        errors["username"] = "Este username já está registado."

    # 3) Email já existe
    if email and User.objects.filter(email=email).exists():
        errors["email"] = "Este email já está registado."

    # 4) Passwords diferentes
    if password1 and password2 and password1 != password2:
        errors["password2"] = "As passwords não coincidem."

    # 5) Regras da password (exemplo: 8 chars, 1 maiúscula, 1 minúscula, 1 digito)
    pwd_errors = []
    if password1:
        if len(password1) < 8:
            pwd_errors.append("Pelo menos 8 caracteres.")
        if not re.search(pattern=r"[A-Z]", password1):
            pwd_errors.append("Pelo menos 1 letra maiúscula.")
        if not re.search(pattern=r"[a-z]", password1):
            pwd_errors.append("Pelo menos 1 letra minúscula.")
        if not re.search(pattern=r"\d", password1):
            pwd_errors.append("Pelo menos 1 número.")

```

Figura 14 – Registo/1

```

if pwd_errors:
    errors["password1"] = ".join(pwd_errors)

# Se houver erros, volta para a mesma página com mensagens
if errors:
    context = {
        "errors": errors,
        "form": form_data,
    }
    return render(request, template_name="sistema_login/registo.html", context)

# Se tudo ok → cria utilizador e redireciona para login
user = User.objects.create_user(username=username, email=email, password=password1)
user.save()
return HttpResponseRedirect("/login/")

```

Figura 15 - Registo/2

O excerto de código apresentado corresponde à view responsável pelo registo de novos utilizadores na aplicação. Esta view trata tanto a apresentação do formulário de registo como o processamento dos dados submetidos pelo utilizador.

Quando é efetuado um pedido do tipo GET, a view limita-se a renderizar a página de registo, disponibilizando o formulário para introdução das credenciais. Em caso de pedido POST, são recolhidos os dados submetidos, nomeadamente o nome de utilizador, o endereço de correio eletrónico e a palavra-passe.

Os dados introduzidos são sujeitos a um conjunto de validações, incluindo a verificação de campos obrigatórios, a unicidade do nome de utilizador e do endereço de correio eletrónico, a correspondência entre as palavras-passe introduzidas e o cumprimento de regras mínimas de segurança relativas à complexidade da palavra-passe. Caso sejam detetados erros, a página de registo é novamente apresentada com as respetivas mensagens de erro, preservando os dados já introduzidos pelo utilizador.

Quando todas as validações são concluídas com sucesso, é criado um novo utilizador no sistema, recorrendo aos mecanismos nativos do Django para a gestão de autenticação. Após a criação do utilizador, este é redirecionado para a página de login, onde poderá autenticar-se com as credenciais definidas.

Esta view assegura um processo de registo seguro e consistente, contribuindo para o correto controlo de acesso à aplicação.

3.3 APIs REST

No âmbito do desenvolvimento da aplicação distribuída, foram implementadas diversas APIs REST, responsáveis pela comunicação entre o backend da aplicação e outros componentes do sistema, nomeadamente os módulos responsáveis pela recolha de dados dos sensores e pelo controlo do estendal inteligente. Estas APIs utilizam o protocolo HTTP e recorrem ao formato JSON para a troca de dados, garantindo interoperabilidade e independência de plataforma.

As APIs desenvolvidas encontram-se organizadas de acordo com as suas funcionalidades principais: sensores, controlo do estendal e emparelhamento do estendal.

3.3.1 API de Sensores

```
@csrf_exempt 由 FranciscoFOnseca
@require_POST
def ingest_sensor(request):
    try:
        data = json.loads(request.body.decode("utf-8"))
    except json.JSONDecodeError:
        return JsonResponse( data: {"ok": False, "error": "JSON inválido"}, status=400)

    serial = (data.get("serial_number") or "").strip().upper()
    if not serial:
        return JsonResponse( data: {"ok": False, "error": "serial_number em falta"}, status=400)

    try:
        rack = DryingRack.objects.get(serial_number=serial)
    except DryingRack.DoesNotExist:
        return JsonResponse( data: {"ok": False, "error": "Estendal não encontrado"}, status=404)

    sensor = Sensor.objects.create(
        drying_rack=rack,
        temperature=data.get("temperature"),
        humidity=data.get("humidity"),
        light_level=data.get("light_level"),
        rain=data.get("rain", False),
        drying_time_estimate=data.get("drying_time_estimate"),
        clothesline_state=data.get(
            "clothesline_state",
            ClotheslineState.EXTENDED,
        ),
    )

    return JsonResponse(
        {
            "ok": True,
            "sensor_id": sensor.id,
        }
    )
```

Figura 16 - API Sensors

A API de Sensores foi desenvolvida para permitir a ingestão e armazenamento das leituras recolhidas pelos sensores associados ao estendal inteligente. Esta API segue o estilo REST, utilizando o protocolo HTTP e o formato JSON para a troca de dados.

A receção dos dados é realizada através de um endpoint que aceita exclusivamente requisições do tipo POST, garantindo que apenas operações de escrita são permitidas. O corpo do pedido contém a informação proveniente dos sensores, a qual é descodificada e validada pelo backend. Caso o formato JSON seja inválido ou existam campos obrigatórios em falta, a API devolve uma resposta de erro apropriada, acompanhada do respetivo código HTTP.

Cada pedido inclui o número de série do estendal, utilizado para identificar de forma única o dispositivo associado. O backend valida este identificador e verifica a existência do estendal na base de dados. Caso o estendal não seja encontrado, é devolvida uma resposta de erro com o código HTTP adequado.

Após a validação dos dados, é criado um novo registo de sensor na base de dados, contendo informações como temperatura, humidade, nível de luminosidade, deteção de chuva, estimativa do tempo de secagem e estado atual do estendal. A persistência dos dados é realizada através do ORM do Django, garantindo uma interação segura e abstrata com a base de dados.

No final do processo, a API devolve uma resposta em formato JSON a indicar o sucesso da operação, incluindo o identificador do registo criado. Esta API constitui um elemento central do sistema distribuído, permitindo a integração entre os dispositivos físicos de recolha de dados e a aplicação web de monitorização.

3.3.2 API de Controlo do Estendal

```
@csrf_exempt @FranciscoFOnseca
@require_POST
@login_required
def control_clothesline(request):
    try:
        data = json.loads(request.body.decode("utf-8"))
    except json.JSONDecodeError:
        return JsonResponse(
            data: {"ok": False, "error": "JSON inválido"},
            status=400
        )

    serial_number = data.get("serial_number")
    action = data.get("action")

    if not serial_number or action not in ["open", "close"]:
        return JsonResponse(
            data: {"ok": False, "error": "Parâmetros inválidos"},
            status=400
        )

    try:
        rack = DryingRack.objects.get(serial_number=serial_number)
    except DryingRack.DoesNotExist:
        return JsonResponse(
            data: {"ok": False, "error": "Estendal não encontrado"},
            status=404
        )

    # Segurança: só o dono pode controlar
    if rack.user != request.user:
        return JsonResponse(
            data: {"ok": False, "error": "Sem permissão para controlar este estendal"},
            status=403
        )

    new_state = "extended" if action == "open" else "retracted"
```

Figura 17 - API Control/1

```

# Criar nova leitura de sensor
sensor = Sensor.objects.create(
    drying_rack=rack,
    datetime=timezone.now(),
    clothesline_state=new_state
)

# Criar alerta manual
Alert.objects.create(
    drying_rack=rack,
    alert_type="manual",
    message=f"Estendal {'aberto' if action == 'open' else 'fechado'} manualmente"
)

return JsonResponse({
    "ok": True,
    "state": new_state,
    "sensor_id": sensor.id
})

```

Figura 18 - API Control/2

A API de Controlo do Estendal foi desenvolvida para permitir a execução de comandos remotos sobre o estendal inteligente, possibilitando a sua abertura e fecho através da aplicação web. Esta API segue o modelo REST, utilizando o protocolo HTTP e o formato JSON para a troca de dados. O endpoint de controlo aceita exclusivamente requisições do tipo POST e encontra-se protegido pelo mecanismo de autenticação do Django, garantindo que apenas utilizadores autenticados podem executar ações sobre o sistema. Adicionalmente, a API implementa um controlo de permissões, assegurando que apenas o proprietário do estendal pode enviar comandos de controlo para o respetivo dispositivo.

Cada pedido inclui o número de série do estendal e a ação a executar, podendo esta corresponder à abertura ou ao fecho do estendal. Os dados recebidos são validados pelo backend, sendo rejeitados pedidos com parâmetros inválidos ou em formato JSON incorreto, através da devolução de respostas de erro adequadas e respetivos códigos HTTP.

Após a validação dos dados e da autorização do utilizador, o backend identifica o estendal correspondente e atualiza o seu estado interno de funcionamento, refletindo a ação solicitada. Este mecanismo permite garantir a integridade do sistema e evita acessos ou comandos não autorizados.

A resposta da API é devolvida em formato JSON, indicando o sucesso ou insucesso da operação. Esta API constitui um componente essencial do sistema distribuído, permitindo a interação segura e controlada entre a interface web e o estendal inteligente.

No seguimento da execução do comando de controlo, o sistema procede ao registo do novo estado do estendal. Para esse efeito, é criada uma nova leitura de sensor associada ao estendal, contendo a data e hora da ação e o respetivo estado do estendal após a execução do comando (aberto ou fechado). Este registo permite manter um histórico consistente das alterações de estado, integrando ações manuais no conjunto de dados monitorizados pelo sistema.

Adicionalmente, é criado um alerta do tipo manual, associado ao estendal, indicando explicitamente a ação realizada pelo utilizador. A mensagem do alerta identifica se o estendal foi aberto ou fechado manualmente, permitindo que esta informação seja posteriormente apresentada na Dashboard e na página de Histórico.

Por fim, a API devolve uma resposta em formato JSON, indicando o sucesso da operação, o novo estado do estendal e o identificador da leitura de sensor criada. Este mecanismo garante a rastreabilidade das ações efetuadas e reforça a consistência entre o controlo do estendal, o registo de sensores e o sistema de eventos, elementos fundamentais numa arquitetura de sistemas distribuídos.

3.3.3 API de Emparelhamento do Estendal

```
@csrf_exempt & FranciscoFOnseca
@require_POST
@login_required
def pair_drying_rack(request):
    try:
        data = json.loads(request.body.decode("utf-8"))
    except json.JSONDecodeError:
        return JsonResponse(
            data: {"ok": False, "error": "JSON inválido"},
            status=400
        )

    serial_number = data.get("serial_number")
    pairing_code = data.get("pairing_code")
    name = data.get("name", "").strip()
    location = data.get("location", "").strip()

    if not serial_number or not pairing_code:
        return JsonResponse(
            data: {"ok": False, "error": "Número de série e código obrigatórios"},
            status=400
        )

    try:
        rack = DryingRack.objects.get(serial_number=serial_number)
    except DryingRack.DoesNotExist:
        return JsonResponse(
            data: {"ok": False, "error": "Estendal não encontrado"},
            status=404
        )

    if rack.pairing_code != pairing_code:
        return JsonResponse(
            data: {"ok": False, "error": "Código de emparelhamento incorreto"},
            status=403
        )
```

Figura 19 - API Pair/1

```

if rack.user and rack.user != request.user:
    return JsonResponse(
        data: {"ok": False, "error": "Este estendal já pertence a outro utilizador"}, 
        status=409
    )

rack.user = request.user
rack.active = True

if name:
    rack.name = name
if location:
    rack.location = location

rack.save()

return JsonResponse({
    "ok": True,
    "rack_id": rack.id,
    "name": rack.name,
    "serial_number": rack.serial_number
})

```

Figura 20 - API Pair/2

A API de Emparelhamento foi desenvolvida com o objetivo de associar de forma segura um estendal inteligente a um utilizador da aplicação. Este processo é fundamental para garantir que apenas utilizadores autorizados conseguem gerir e controlar um determinado dispositivo.

O endpoint de emparelhamento aceita exclusivamente requisições do tipo POST e encontra-se protegido pelos mecanismos de autenticação do Django, garantindo que apenas utilizadores autenticados podem iniciar o processo de emparelhamento. Os dados são enviados em formato JSON, sendo devidamente validados pelo backend.

Cada pedido de emparelhamento inclui o número de série do estendal e um código de emparelhamento, bem como informação adicional opcional, como o nome e a localização do dispositivo. O backend valida a presença dos campos obrigatórios e verifica a existência do estendal na base de dados.

Após a identificação do estendal, o sistema valida o código de emparelhamento, assegurando que apenas utilizadores na posse de um código válido podem concluir o processo. Caso o código seja incorreto ou o estendal não exista, a API devolve uma resposta de erro adequada, com o respetivo código HTTP.

Quando todas as validações são bem-sucedidas, o estendal é associado ao utilizador autenticado, ficando registadas as informações adicionais fornecidas. Este mecanismo garante um processo de emparelhamento seguro e controlado, evitando associações indevidas entre dispositivos e utilizadores. A API devolve uma resposta em formato JSON indicando o sucesso da operação. Esta funcionalidade constitui um elemento essencial da arquitetura distribuída do sistema, permitindo a gestão segura de dispositivos físicos através da aplicação web.

3.4 DESAFIOS E SOLUÇÕES

Durante o desenvolvimento do projeto foram encontrados diversos desafios, principalmente relacionados com a criação e utilização de APIs, a integração entre os diferentes componentes do sistema e a configuração do ambiente Docker.

Criação e Consumo de APIs

Um dos principais desafios prendeu-se com a definição e implementação das APIs REST, responsáveis pela comunicação entre o backend Django, o sistema do estendal inteligente e serviços externos. Foi necessário definir corretamente os endpoints, os métodos HTTP a utilizar e o formato dos dados trocados, de modo a garantir uma comunicação consistente e fiável.

Solução:

Este desafio foi resolvido através da utilização do Django REST Framework, que permitiu estruturar as APIs de forma clara, definir serializers para validação dos dados e garantir respostas normalizadas em formato JSON.

Validação e Consistência dos Dados

Outro desafio identificado foi a validação dos dados recebidos, especialmente no caso dos dados provenientes dos sensores e das simulações realizadas via API. Dados incompletos ou mal formatados poderiam comprometer o correto funcionamento do sistema.

Solução:

Foram implementados mecanismos de validação no backend, recorrendo a serializers e validações adicionais no Django, garantindo que apenas dados válidos são processados e armazenados na base de dados.

Comunicação entre Containers Docker

A configuração da comunicação entre os containers Docker do backend e da base de dados constituiu também um desafio inicial, nomeadamente ao nível da configuração de redes, portas e variáveis de ambiente.

Solução:

Este problema foi resolvido através da utilização de um ficheiro `docker-compose.yml`, onde foram definidos os serviços, a rede interna e as credenciais necessárias para permitir a comunicação correta entre os containers.

Testes sem Dependência do Hardware

A inexistência permanente do hardware físico do estendal durante o desenvolvimento dificultou a validação contínua do sistema.

Solução:

Este desafio foi ultrapassado através da utilização do Postman, permitindo simular o envio de dados dos sensores e comandos de controlo através de pedidos HTTP em formato JSON, garantindo que o sistema pudesse ser testado de forma independente do hardware.

Integração com Serviços Externos

A integração com a API externa do IPMA apresentou desafios relacionados com a gestão de pedidos HTTP e o tratamento de possíveis erros de comunicação.

Solução:

Foram implementados mecanismos de tratamento de exceções e validação das respostas da API, garantindo que a aplicação se mantém funcional mesmo em situações de indisponibilidade temporária do serviço externo.

Considerações Finais

A superação destes desafios contribuiu para um melhor entendimento dos conceitos de sistemas distribuídos, comunicação cliente-servidor e integração de serviços, permitindo o desenvolvimento de uma aplicação mais robusta e fiável.

4- TESTES E RESULTADOS

4.1 DESCRIÇÃO DOS TESTES

De forma a validar o correto funcionamento do sistema desenvolvido, foram realizados testes unitários e testes de integração.

Testes Unitários:

Os testes unitários tiveram como objetivo validar o comportamento dos componentes isolados do sistema, nomeadamente:

- Validação do sistema de registo e autenticação de utilizadores;
- Validação da criação e armazenamento de eventos na base de dados;
- Verificação do correto processamento de dados provenientes dos sensores;
- Validação de regras de negócio, como o registo automático de eventos em situações específicas

Estes testes permitiram confirmar que cada funcionalidade individual funciona corretamente de forma independente.

Testes de Integração:

Os testes de integração foram realizados para validar a interação entre os diferentes componentes do sistema, nomeadamente:

- Comunicação entre o frontend (browser) e o backend Django através de pedidos HTTP;
- Comunicação entre o backend Django e a base de dados PostgreSQL, garantindo a correta persistência e recuperação dos dados;
- Integração com APIs externas, como a API do IPMA, para obtenção de dados meteorológicos;
- Integração entre o sistema web e o estendal inteligente, verificando o envio e receção de comandos e estados.

Estes testes permitiram validar o funcionamento do sistema como um todo, assegurando que os diferentes módulos comunicam corretamente entre si.

Exemplos de Inputs e Outputs:

Durante os testes, foram utilizados diversos cenários de entrada e observadas as respetivas saídas do sistema. Alguns exemplos incluem:

- **Input:** Credenciais válidas de utilizador no formulário de login
Output: Autenticação bem-sucedida e redirecionamento para a dashboard
- **Input:** Pedido para abrir o estendal manualmente
Output: Atualização do estado do estendal e registo do evento na base de dados
- **Input:** Simulação de deteção de chuva
Output: Fecho automático do estendal e criação de evento correspondente no histórico
- **Input:** Pedido para consulta do histórico das últimas 24 horas
Output: Apresentação dos gráficos de temperatura, humidade e vento, bem como a lista de eventos recentes

Outros Testes de Integração:

Para além dos testes realizados através da interface web, foram também efetuados testes de integração às APIs desenvolvidas, recorrendo à ferramenta Postman.

Através do Postman, foi possível simular o comportamento dos sensores, o processo de emparelhamento (pairing) e o controlo do estendal, sem necessidade de interação direta com o hardware físico. Para tal, foram criados ficheiros no formato JSON, contendo dados representativos dos valores dos sensores e comandos de controlo.

Estes ficheiros JSON foram enviados para os respetivos endpoints das APIs REST, sendo posteriormente processados pelo backend Django e armazenados na base de dados. Este procedimento permitiu validar:

- A correta receção e validação dos dados enviados;
- O processamento da informação no backend;
- A persistência correta dos dados na base de dados;
- A criação automática de eventos associados às ações simuladas.

Os testes realizados confirmaram que as APIs respondem corretamente aos pedidos efetuados, garantindo a fiabilidade da comunicação entre os diferentes componentes do sistema.

4.2 RESULTADOS OBTIDOS

Os testes realizados permitiram confirmar que a aplicação funciona corretamente em ambiente Docker, apresentando uma interface web estável e funcional. Foram capturados printscreens da aplicação em funcionamento, evidenciando as principais páginas do sistema, como a dashboard, o histórico de eventos e a página de definições.

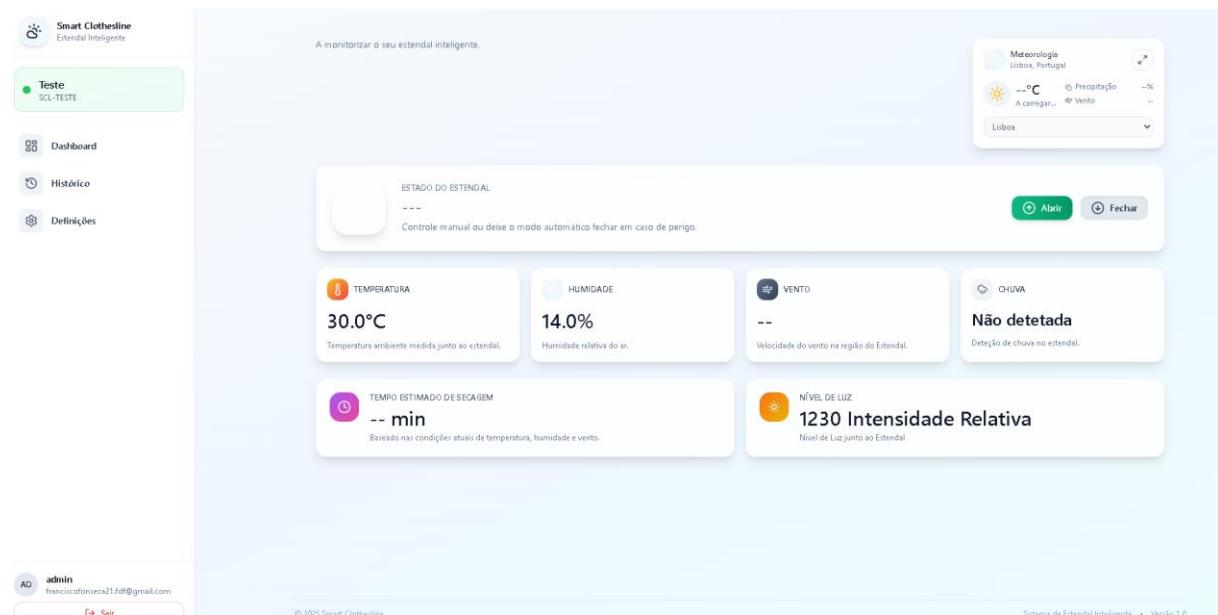


Figura 21 - Dashboard

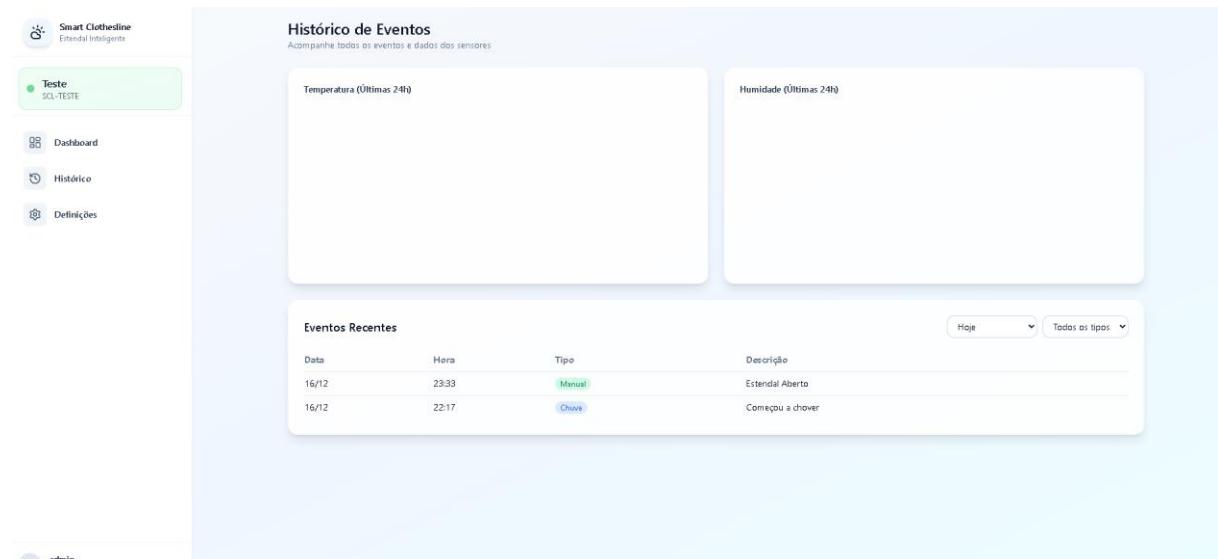


Figura 22 - Histórico

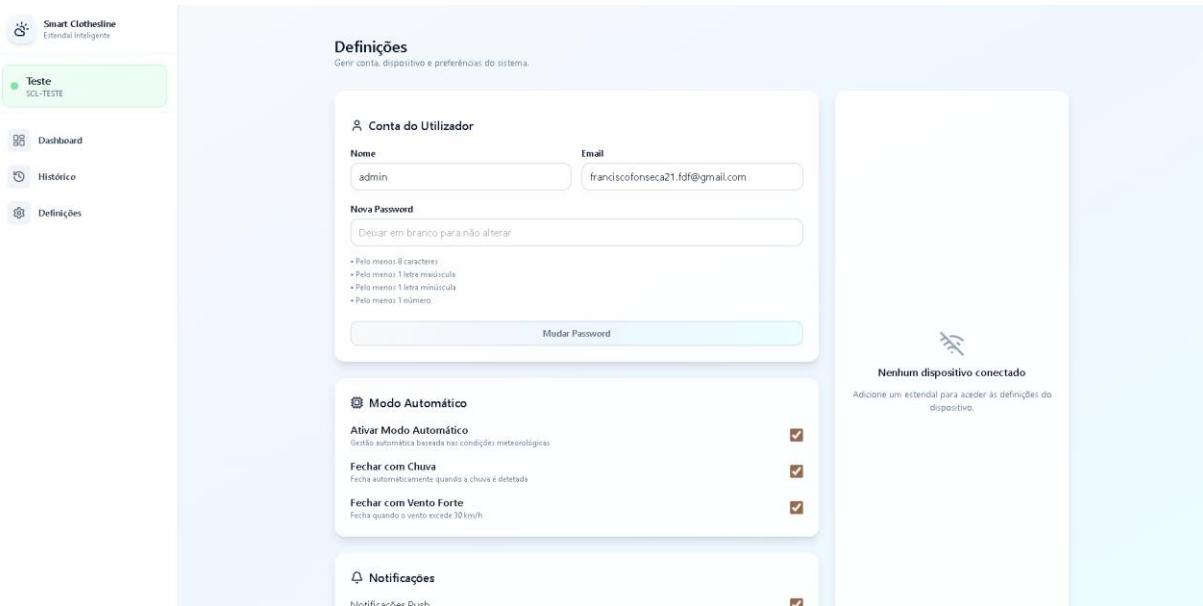


Figura 23 - Definições

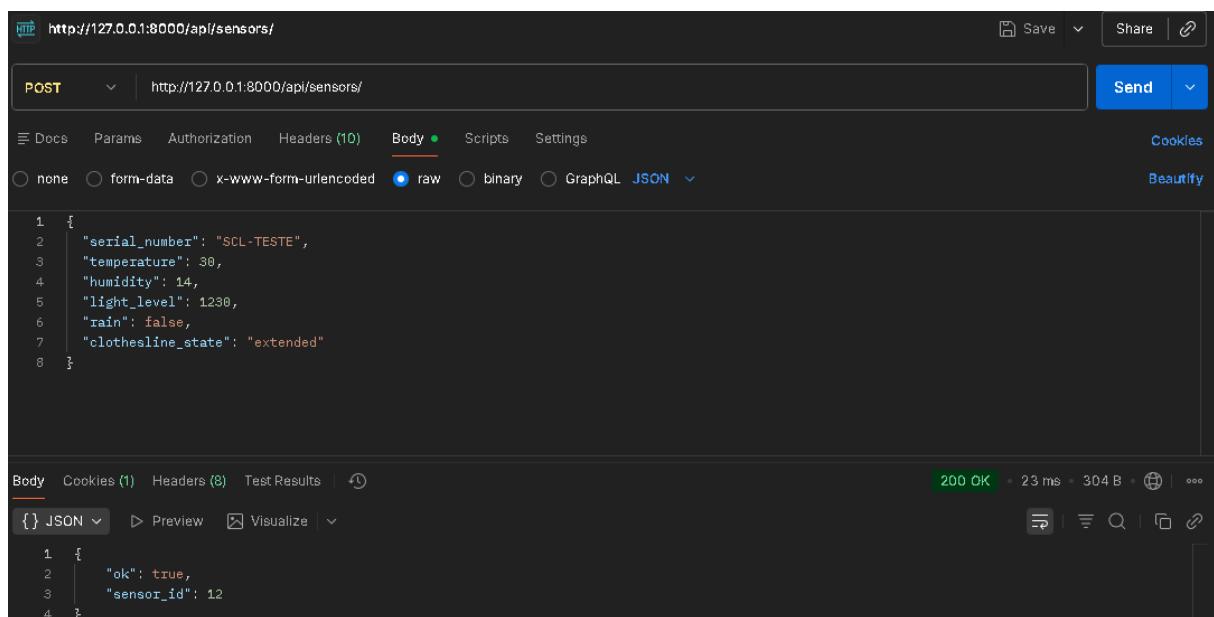


Figura 24 - Teste de API Sensors

The screenshot shows a Postman interface with the following details:

- URL: `http://127.0.0.1:8000/api/pair/`
- Method: `POST`
- Headers:
 - `Content-Type`: `application/json`
- Body (JSON):

```
1 {  
2     "ok": true,  
3     "rack_id": 3,  
4     "name": "Teste",  
5     "serial_number": "SCL-TESTE"  
6 }
```
- Response status: `200 OK`
- Response time: `24 ms`
- Response size: `362 B`

Figura 25 - Teste de API Pair

The screenshot shows a Postman interface with the following details:

- URL: `http://127.0.0.1:8000/api/control/`
- Method: `POST`
- Headers:
 - `Content-Type`: `application/json`
- Body (JSON):

```
1 {  
2     "ok": true,  
3     "state": "extended",  
4     "sensor_id": 14  
5 }
```
- Response status: `200 OK`
- Response time: `41 ms`
- Response size: `339 B`

Figura 26 - Teste de API Control

CONCLUSÃO

Ao longo do projeto foi possível consolidar conhecimentos relacionados com a separação de responsabilidades, nomeadamente entre a camada de apresentação, a lógica de negócio e a persistência de dados, recorrendo ao framework Django e ao seu modelo arquitetural MVT. A utilização de APIs REST revelou-se fundamental para assegurar uma comunicação eficiente, padronizada e independente de plataforma entre os diferentes componentes do sistema, incluindo a integração com dispositivos externos e serviços de terceiros.

No que diz respeito ao deployment, a adoção da tecnologia Docker constituiu um dos aspectos mais relevantes do trabalho. A utilização de containers permitiu criar um ambiente de execução isolado, consistente e facilmente reproduzível, simplificando significativamente o processo de instalação, configuração e execução da aplicação. A definição de múltiplos containers, um dedicado ao backend da aplicação e outro à base de dados, reforçou os princípios de modularidade, escalabilidade e manutenção, características essenciais em sistemas distribuídos modernos. O recurso ao Docker Compose facilitou ainda a orquestração dos serviços e a gestão da comunicação entre os diferentes componentes do sistema.

A abordagem adotada evidencia a importância do deployment automatizado e da contênerização como soluções amplamente utilizadas em ambientes profissionais, permitindo que aplicações distribuídas sejam facilmente escaladas, atualizadas ou migradas entre diferentes infraestruturas, quer em ambientes locais, quer em plataformas de cloud computing.

Em contexto de vida real, os conhecimentos adquiridos neste trabalho são diretamente aplicáveis em diversos cenários, como o desenvolvimento de aplicações web escaláveis, sistemas de Internet of Things (IoT), plataformas de monitorização remota, serviços cloud e arquiteturas baseadas em microserviços. A capacidade de desenvolver aplicações distribuídas, definir APIs bem estruturadas, integrar serviços externos e realizar deployment recorrendo a containers constitui uma competência essencial no mercado de trabalho atual.

BIBLIOGRAFIA

https://www.djangoproject.com/en/2.2/ref/settings/#std:setting-SECRET_KEY

<https://docs.djangoproject.com/en/2.2/topics/auth/default/>

<https://tailwindcss.com/>

<https://www.postgresql.org/docs/12/>

<https://learning.postman.com/docs/introduction/overview/>