ISEL – Instituto Superior de Engenharia de Lisboa
DEETC – Departamento de Engenharia de Electrónica e Telecomunicações e de Computadores

LEIM
Licenciatura em Engenharia Informática e Multimédia
Unidade Curricular de Projeto

Desenvolvimento de estação meteorológica – Módulo de sensores

Diogo Ribeiro (46307@alunos.isel.ipl.pt)

Orientadores

Professor Doutor Carlos Gonçalves

Professor Doutor Pedro Miguens

setembro, 2021

Resumo

O projeto apresentado neste relatório consiste na implementação de um serviço que potencie uma rede colaborativa de estações meteorológicas, capaz de recolher os seus dados, calcular previsões meteorológicas e mostrar um boletim meteorológico.

O foco da solução é simplificar sistemas meteorológicos para casos locais, suportar estações meteorológicas amadoras e integração com dispositivos IoT.

O sistema está dividido em quatro módulos: uma estação exemplo, uma base de dados meteorológicos, o serviço de previsão meteorológica e o servidor *Web*.

O resultado do projeto desenvolvido é um serviço flexível que completa uma interligação entre sensores e páginas *Web* com foco na ergonomia e compatibilidade, pronto para escalar.

Palavras-chave

Sensores meteorológicos; Integração IoT; Previsão numérica do tempo; Visualização de dados; Sistema de Informação Geográfica; Rede colaborativa; Software-como-um-serviço.

Abstract

The project presented in this report consists on implementing a service that leverages a collaborative network of weather stations, able to collect their data, calculate weather forecasts and display a weather briefing.

The aim of the solution is to simplify weather systems for local cases, to support amateur weather stations and integration with IoT devices.

The system is separated into four modules: an example station, a weather database, the weather forecast service and the *Web* server.

The outcome of the developed project is a flexible service that provides a seamless interconnection from sensor to web page with a focus on ergonomics and compatibility, ready to scale.

Keywords

Weather sensors; IoT integration; Numerical weather prediction; Data visualization; Geographical Information System; Collaborative network; Software-as-a-service.

Agradecimentos

Quero agradecer ao Instituto Superior de Engenharia de Lisboa, em particular ao Departamento de Engenharia Eletrónica, Telecomunicações e de Computadores, pelos meios que colocou à disposição para a minha formação.

Aos professores Carlos Gonçalves e Pedro Miguens, orientadores deste projecto, pela grande disponibilidade e afabilidade que manifestaram desde início e pelo apoio prestado.

Gostaria também de agradecer ao grupo de amigos que fiz durante a faculdade por me terem acompanhado e colaborado comigo ao longo destes três anos para ultrapassar qualquer obstáculo que nos fosse imposto.

Finalmente, gostaria de agradecer ao meu pai que tem trabalhado arduamente para me permitir chegar a onde estou.

No seguimento deste trabalho gostaria de prestar homenagem ao meu tio-avô Vítor, que perdeu a batalha contra o cancro faz três anos.

Ele foi a pessoa responsável por introduzir ao meu pai às novas formas de tecnologia da altura e à engenharia no ISEL.

Consequentemente, e também através da minha experiência direta com ele, inspirou-me a ser quem sou e quem quero ser.

Índice de Conteúdos

Índice de Figuras

Índice de Tabelas

Lista de Equações

Lista de Acrónimos

%RH Humidade relativa. 1

AJAX *Asynchronous JavaScript and XML*. 1, 51, 76

API *Application Programming Interface*. 1, 32, 43, 54

AR Auto-regressivo. 1, 5

ARIMA *Autoregressive integrated moving average*. 1, 60

AROME *Application of research at the operational mesoscale*. 1, 4

CRUD *Create, Read, Update & Delete*. 1, 43

CSS *Cascading Style Sheets*. 1, 42, 48, 49

CSV *Comma-Separated Values*. 1, 60

DEETC Departamento de Engenharia Electrónica e Telecomunicação e de Computadores. 1

deg Grau (geometria). 1

DNS *Domain Name System*. 1, 73

EA Entidade-Associação. xiii, 1, 16, 17

FTP *File Transfer Protocol*. 1, 60

GPS *Global Positioning System*. 1, 76

hPa Hectopascal. 1

HTML *Hypertext Markup Language*. 1, 20, 42, 47, 48

HTTP *Hypertext Transfer Protocol*. 1, 9, 26, 32, 49–51, 68

I2C *Inter-Integrated Circuit*. 1, 25, 32, 33, 63

IoT Internet das Coisas (do inglês *Internet of Things*). i, 1, 12, 28, 32, 77

IP Internet Protocol. 1, 21, 51, 68, 73, 76

IPL Instituto Politécnico de Lisboa. 1, 66

IPMA Instituto Português do Mar e da Atmosfera. 1, 4

ISEL Instituto Superior de Engenharia de Lisboa. 1, 66

JSON *JavaScript Object Notation*. 1, 18, 31–33, 38, 40, 46, 48, 51, 76

km/h Quilómetros por hora. 1

mL Mililitro. 1

MQTT *Message Queuing Telemetry Transport*. xiv, 1, 25, 26, 32, 34, 37, 46, 58, 64, 65, 76

NAT *Network Address Translation*. 1, 72, 73

NTP *Network Time Protocol*. 1, 37

NWP Previsão numérica do clima (do inglês *Numerical Weather Prediction*). 1, 6

PCB *Printed Circuit Board*. 1, 28

PSRAM Pseudo *Static Random Access Memory*. 1, 28

QOL *Quality of Life*. 1, 76

QoS *Quality of Service*. 1, 34, 35, 38, 64

RAM Random Access Memory. xx, xxi, 1, 28

REPL *Read Evaluate Print Loop*. xiv, 1, 29, 30, 63, 64

REST *Representational state transfer*. 1, 32

RF Requisitos funcionais. 1, 10

RM *Regression model with correlated errors*. 1, 5

RNF Requisitos não-funcionais. 1, 10, 11

SaaS Software-como-um-serviço (do inglês *Software-as-a-service*). 1

SMTP *Simple Mail Transfer Protocol*. 1, 26

SoC *Series of Chips*. 1, 27

SQL *Structured Query Language*. 1

SRAM *Static Random Access Memory*. xx, 1, 75

SSL *Secure Sockets Layer*. 1, 32

SVG *Scalable Vector Graphics*. 1, 66

TBATS *Exponential smoothing state space model with Box-Cox transformation, ARMA errors, Trend and Seasonal components*. 1, 5

TLS *Transport Layer Security*. 1, 32

UI *User Interface*. 1

UML *Unified Modeling Language*. xiv, 1, 33

URL *Universal Resource Locator*. 1, 49

Wi-Fi Protocolos de Rede *Wireless* IEEE 802.11. 1, 27, 28, 63

WLAN *Wireless Local Area Network*. 1, 32, 37

WSGI *Web Server Gateway Interface*. 1, 76

XML *EXtensible Markup Language*. xix, 1, 32, 40

YAML *Yet Another Markup Language*. 1, 31

°C Graus Celsius. 1

Capítulo 1

Introdução

Inspirado no atual paradigma da Internet das Coisas, este projeto tem como motivação tornar os complexos sistemas de ciência meteorológica em algo acessível e fácil de implementar.

Para tal, será desenvolvido um serviço de suporte a uma rede colaborativa de estações meteorológicas. Cada estação meteorológica é constituída por um microcontrolador, um módulo de comunicação e um conjunto de módulos de sensores que enviam os dados recolhidos para um servidor.

Atualmente, a maior parte dos programas de processamento de dados de uma estação meteorológica requerem modelos específicos e estações pré-construídas. Para além disso, são habitualmente adquiridas em separado e a fornecedores diferentes. Este trabalho procura facilitar um cenário em que, com recurso a estações domésticas, são monitorizadas de perto as condições de uma pequena área ou microclima. Para tal é solicitado ao utilizador o registo da sua estação no serviço aqui proposto e desenvolvido. À estação é atribuída um endereço e instruções para o envio dos dados.

Depois de preparado, o serviço periodicamente (a definir pelo utilizador) recolhe os dados e realiza todos os cálculos. Entre estes cálculos estão incluídas as previsões de curto prazo através de modelos matemáticos, que podem ajudar a antecipar circunstâncias como seca ou frio excessivo para, por exemplo, horticultura. Os resultados são disponibilizados pelo serviço através de um servidor *Web* sob a forma de boletim meteorológico em páginas acessíveis ao público.

Este projeto implementa também a notificação e regulação das condições meteorológicas registadas pelo sistema, sendo que o servidor permite o envio

de avisos por correio eletrónico a utilizadores registados, desde que estes especifiquem a condição. Em versões futuras do sistema proposto poderia ser também acompanhado por uma aplicação móvel ou integrado com os sistemas de climatização ou rega em si.

1.1 Organização do documento

No Capítulo 2 descreve-se e analisa-se o trabalho relacionado. No Capítulo 3, apresenta-se a arquitetura do sistema proposto e requisitos. A implementação do sistema é descrita no Capítulo 4. A validação, testes e resultados experimentais são descritos no Capítulo 5. Conclui-se com um balanço do trabalho realizado e sugestões de trabalho futuro no Capítulo 6.

O dossiê do projeto, incluindo *datasheets* e código fonte, está situado no seguinte repositório no GitHub: [1].

Capítulo 2

Trabalho Relacionado

Ao planear o projeto foram realizadas pesquisas de forma a determinar o estado da arte das tecnologias envolvidas. Vários projetos com caraterísticas em comum foram considerados de forma a definir robustamente os requisitos do projeto.

Um exemplo de interligação entre uma estação meteorológica e a Internet é o Meteobridge [2], um projeto com o objetivo de agregar várias estações através da sua ligação com um *router* dedicado a rede meteorológicas públicas. Esta solução tem custos: o software é *shareware* que requere uma licença ao fim de 14 dias de uso e o hardware requere que seja utilizado um de quatro modelos de *router* específicos. A Figura 2.1 ilustra a arquitetura do sistema em questão.

Figura 2.1: Ilustração de um sistema com Meteobridge instalado
Fonte: [2]

Outro projeto nesta área é o Cumulus [3], que tem como objetivo recolher, armazenar e apresentar dados de estações meteorológicas comerciais (depende de modelos específicos). Este projeto já tem várias iterações, sendo

que a atual explicita a implementação de um servidor *Web* para transferência de ficheiros a partir da estação meteorológica. É distribuído como *donationware*, que ao contrário de *freeware* tem pedidos frequentes por doações dos utilizadores. A Figura 2.2 apresenta o ecrã da aplicação Windows da primeira versão do projeto Cumulus.

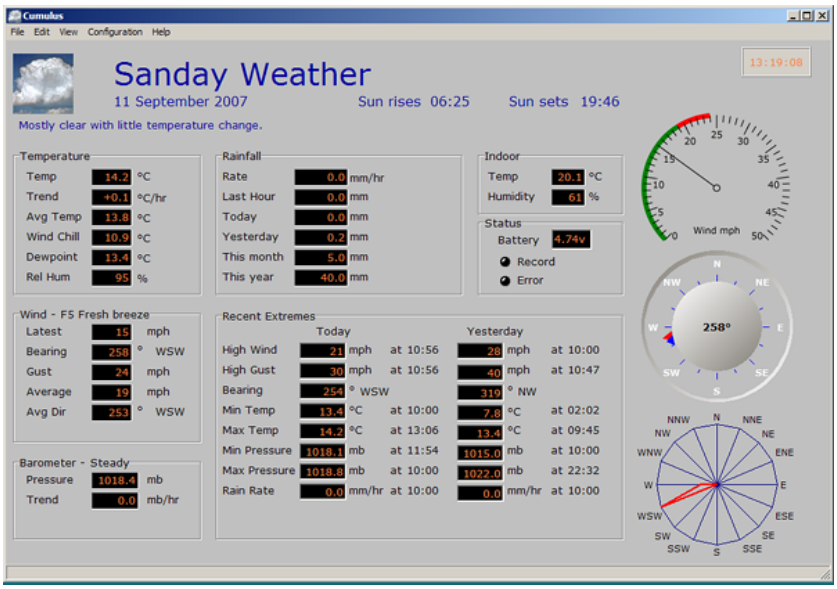

Figura 2.2: Interface gráfica do Cumulus 1
Fonte: [3]

As páginas do Instituto Português do Mar e da Atmosfera [4] e MeteoMoita.com [5] são exemplos de páginas que apresentam o estado presente e previsão do clima, no primeiro caso aplicado a todo o país e no segundo apenas a uma estação na Moita. Ao contrário do trabalho desenvolvido, estas páginas dependem de modelos atmosféricos como AROME [6] e outros no caso do IPMA e VXSIM [7] no caso do MeteoMoita para previsão do tempo. A Figura 2.3 apresenta o aspeto da página inicial do serviço MeteoMoita, que inclui um boletim meteorológico para o município da Moita.

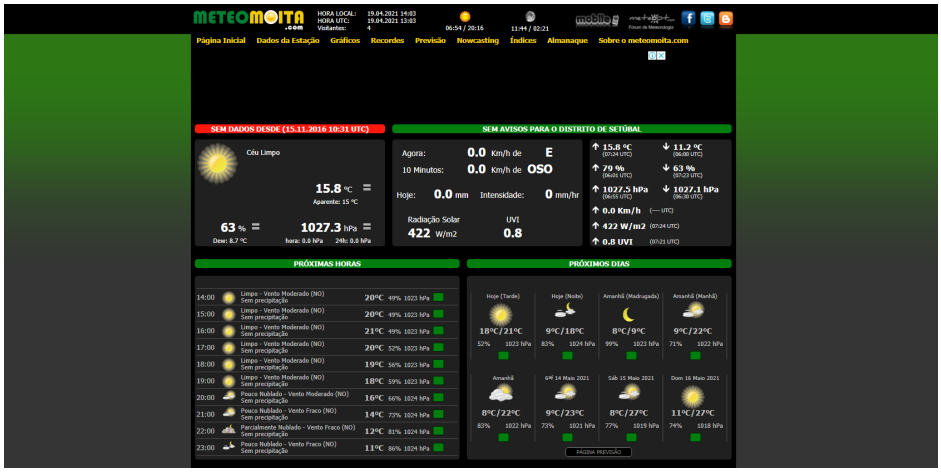

Figura 2.3: Página inicial do MeteoMoita.com
Fonte: [5]

2.1 Modelo estatístico de previsão do clima

Quanto à previsão do clima, foram analisados diferentes artigos que explicam o uso de modelos matemáticos de série cronológica de forma a simular dados futuros a partir de tendências cíclicas.

Exemplos destes incluem *"Temperature time series forecasting in the optimal challenges in irrigation"* [8], que em resposta às épocas de seca em Portugal explora o uso de modelos *Regression model with correlated errors* (RM) e *Exponential smoothing state space model with Box-Cox transformation, ARMA errors, Trend and Seasonal components* (TBATS) da linguagem R e conclui que estes capturam corretamente as variações de temperatura mínima do ar observados de 2015 a 2019 para uma única estação no local escolhido e para avaliação dos modelos prevêm, com 2015 a 2018 como conjunto de treino, a temperatura mínima ao longo do ano de 2019.

Para além do exemplo anterior, *"Machine Learning in Python for Weather Forecast based on Freely Available Weather Data"* [9] é um artigo que descreve a aplicação da biblioteca TensorFlow em Python para previsão da temperatura a partir de modelos Auto-regressivo (AR). No artigo em questão, quatro modelos separados foram treinados para prever a temperatura com 1, 3, 6, e 12 horas de antecedência. Na primeira experiência, foi utilizada apenas a temperatura como dados de entrada para as redes, enquanto na

segunda experiência, foram introduzidos, para além dos dados de temperatura, dados da precipitação. No artigo é descrito que, após uma extensa calibração de hiperparâmetros, os resultados da previsão dos modelos foram comparados, e concluiu-se que a introdução da precipitação como entrada no modelo aparenta melhorar ligeiramente o desempenho da previsão. Assim, concluiu-se que poderia ser interessante acrescentar mais dimensões de entrada ao modelo.

Finalmente, o artigo *"Statistical Single-Station Short-Term Forecasting of Temperature and Probability of Precipitation: Area Interpolation and NWP Combination"* [10] apresenta uma perspectiva estatística como alternativa a modelos Previsão numérica do clima (do inglês *Numerical Weather Prediction*) (NWP), para previsão de temperatura e probabilidade de precipitação numa única estação. Utilizando modelos R (Regressão) e M (Markov), e algumas técnicas simples de interpolação, o trabalho apresenta previsões de característica regional com os dados obtidos. A comparação com modelos estatísticos simples como a média climática mostra uma melhoria da precisão obtida pelos modelos estatísticos mais avançados. Como o desempenho dos modelos estatísticos mostra bons resultados, a combinação linear destes modelos com previsões meteorológicas numéricas alcança melhorias adicionais na precisão da previsão.

O desafio de previsões para uma estação em ciências meteorológicas é definido pela dimensão temporal e espacial das mesmas.

Desde logo as caraterísticas mais adequadas ao projeto foram identificadas como espaço local e curto prazo. Espaço local porque foi previsto que o módulo de sensores fosse uma peça conjunta e como tal obtém dados de uma área restrita. As previsões são independentes entre estações, apesar de uma previsão conjunta ser potencial trabalho para o futuro do sistema aqui proposto.

Com preocupação na vertente científica do projeto, foi enviada uma mensagem de correio eletrónico ao investigador Stephan Siemen, com a finalidade de perguntar como incorporar neste projeto, de forma correta, a previsão meteorológica. O investigador tinha antes sido responsável por várias palestras acerca do uso de *Python* para previsões meteorológicas. A resposta que recebi foi esclarecedora:

> *"The complexity of setting up a full forecast model might exceed*

> *your time limit. To do a very short weather forecast with a single weather stations you could train a regression model. This model could be used to make short time forecast. To train any model like that would require very long time series of data."*
> — Stephen Siemen (restante correspondência no Apêndice A)

Em conclusão, opta-se por um modelo matemático *"série cronológica"* (*time-series*) analisando apenas as tendências em ciclos e aceitando as consequências que resultam da falta de simulação atmosférica e fusão de dados ao nível regional, nacional e europeu.

Capítulo 3

Modelo Proposto

A raíz do sistema é definida por um serviço meteorológico, que com apoio de um sistema de base de dados e um servidor HTTP serve páginas *Web* a visitantes. Por detrás deste serviço existe uma rede de servidores de comunicação dedicados a servir como intermediários de mensagens entre o serviço e estações meteorológicas. Estas estações enviam dados extraídos dos seus sensores.

É da responsabilidade do utilizador desenvolver a sua própria estação meteorológica; no entanto, no caso do projeto desenvolvido é prototipada uma estação como exemplo.

A Figura 3.1 representa as partes do sistema desenvolvido e interação entre estas de um ponto de vista de alto nível.

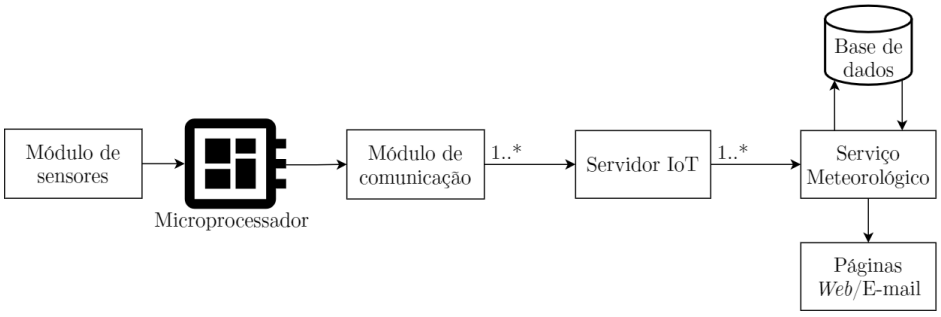

Figura 3.1: Arquitetura simplificada do projeto

No presente projeto o processo de desenvolvimento baseia-se num modelo em cascata (do inglês, *Waterfall*). Este é um modelo de desenvolvimento de software linear e sequencial, popularizado por W. W. Royce em 1970 [11],

que tem como vantagens implicar menos alterações depois do planeamento, bem como não necessitar de treino prévio, o que o torna o mais indicado para projetos curtos e com pouca mão de obra. Por outro lado, as desvantagens deste modelo incluem a sua rigidez, predisposição a atrasos nos prazos e inibição da criatividade.

Neste capítulo é descrito o modelo proposto para o projeto, nomeadamente o seu planeamento através de análise de requisitos na Secção 3.1 e artefactos na Secção 3.2, casos de utilização na Secção 3.3 e finalmente modelo de dados na Secção 3.4.

3.1 Requisitos

Os Requisitos funcionais (RF) e Requisitos não-funcionais (RNF) distinguem-se facilmente pelo uso de duas expressões: um RF é *o que o sistema deve fazer* enquanto que um RNF é *como o sistema deve fazer*.

Na Secção 3.1.1 apresenta-se a análise dos requisitos funcionais, seguida dos requisitos não funcionais na Secção 3.1.2.

3.1.1 Requisitos funcionais

Os Requisitos funcionais representam as funções do sistema e podem ser categorizados como evidentes, invisíveis ou ornamentais.

Dos requisitos analisados na Tabela 3.1, destaca-se a invisibilidade das funções relativas à leitura, envio e tratamento de dados, que são depois da sua configuração realizadas periodicamente sem preocupação da parte do utilizador. Realça-se também a categoria de adorno que a representação gráfica dos dados toma, dado que sem esta função os dados podem ser apresentados na sua forma numérica.

Tabela 3.1: Funções do sistema

Ref.	***Função***	***Categoria***
RF1	Leitura e interpretação dos sensores meteorológicos	Invisível
RF2	Envio dos dados para o servidor	Invisível
RF3	Limpeza e análise dos dados recebidos dos sensores	Evidente
RF4	Recolha e tratamento de dados meteorológicos históricos	Invisível
RF5	Previsões meteorológicas recentes no tempo e locais no espaço	Evidente
RF6	Representação gráfica dos dados históricos acumulados	Adorno

3.1.2 Requisitos não funcionais

Os Requisitos não-funcionais representam atributos do sistema e a sua prioridade, podendo ser classificados por duas categorias: obrigatórios ou desejáveis.

Na Tabela 3.2 constam os atributos considerados para o sistema, sendo que não é crítico, mas desejável, que as suas previsões meteorológicas sejam idênticas às do método científico e que as páginas onde a informação é apresentada sigam todas as regras da acessibilidade.

Tabela 3.2: Atributos do sistema

Ref.	Atributo	Detalhe	Categoria
RNF1	Fidedigno	Ênfase na robustez das mensagens	Obrigatório
		Deve estar disponível para receber os dados do servidor sempre que possível	Obrigatório
RNF2	Interoperável	Desenvolvimento de um servidor *Web* com base de dados de suporte	Obrigatório
RNF3	Flexível	Implementação do máximo de unidades de medição possíveis	Desejável
RNF4	Plausível	Preocupação com a aproximação a um modelo científico	Desejável
RNF5	Acessível	Disponibilização da informação recolhida em páginas *Web*	Desejável

3.2 Artefactos

O projeto agrega um conjunto de artefactos, sendo estes os objetivos analisados na Secção 3.2.1, o público-alvo na Secção 3.2.2 e finalmente metas a alcançar na Secção 3.2.3.

3.2.1 Síntese de objetivos

Neste projeto será desenvolvida uma interface dedicada a concretizar a interligação entre diversos sensores meteorológicos e interpretação os seus dados. O sistema deverá ser extensível a qualquer conjunto de sensores e actuadores, através da comunicação destes sobre um formato de troca de dados estruturado com um serviço abstrato.

3.2.2 Clientes

O projeto desenvolvido dirige-se a utilizadores relacionados a agricultura, floricultura e silvicultura, no sentido em que tencionem monitorizar as condições meteorológicas nas suas hortas, estufas ou jardins. Não se limitando ao setor industrial, pode também existir intenção em monitorizar locais de interesse meteorológico, nomeadamente micro-climas.

3.2.3 Metas a alcançar

Pretende-se que este sistema contribua para:

- Colaboração entre estações meteorológicas;
- Maior precisão espacial de previsões meteorológicas;
- Fácil integração com outros sistemas IoT.

3.3 Casos de utilização

Um diagrama de casos de utilização especifica, a partir de uma visão de alto nível, a interação entre atores e funcionalidades do sistema, expondo assim o valor que o sistema tem para o utilizador. Neste sentido, foram preparados três diagramas de casos de utilização, cada um focando-se nos três níveis do trabalho em desenvolvimento.

Para o caso da recolha de dados, o programa em tempo de execução está encarregue de ler dados do sensor e marcar a coleção destes com marca horária, enviando depois a mensagem para um remetente que a envia para um servidor. Esta interação encontra-se demonstrada na Figura 3.2.

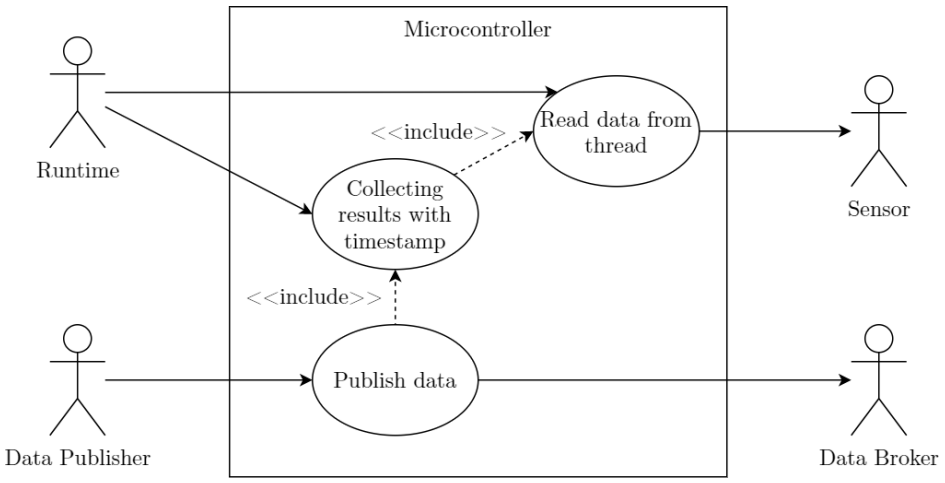

Figura 3.2: Caso de utilização - Recolha de dados

Avançando para a interação dentro do servidor, denota-se na Figura 3.3 que o sistema depende de um cliente destinatário que esteja à escuta por novas mensagens no servidor vindas do remetente representado na Figura 3.2. Ao detetar uma nova mensagem, o destinatário recebe os dados nesta incluídos. Esta ação desencadeia a limpeza e armazenamento destes na base de dados. Finalmente, o servidor processa os dados armazenados para gerar boletins e previsões meteorológicas com auxílio de uma biblioteca de modelos estatísticos.

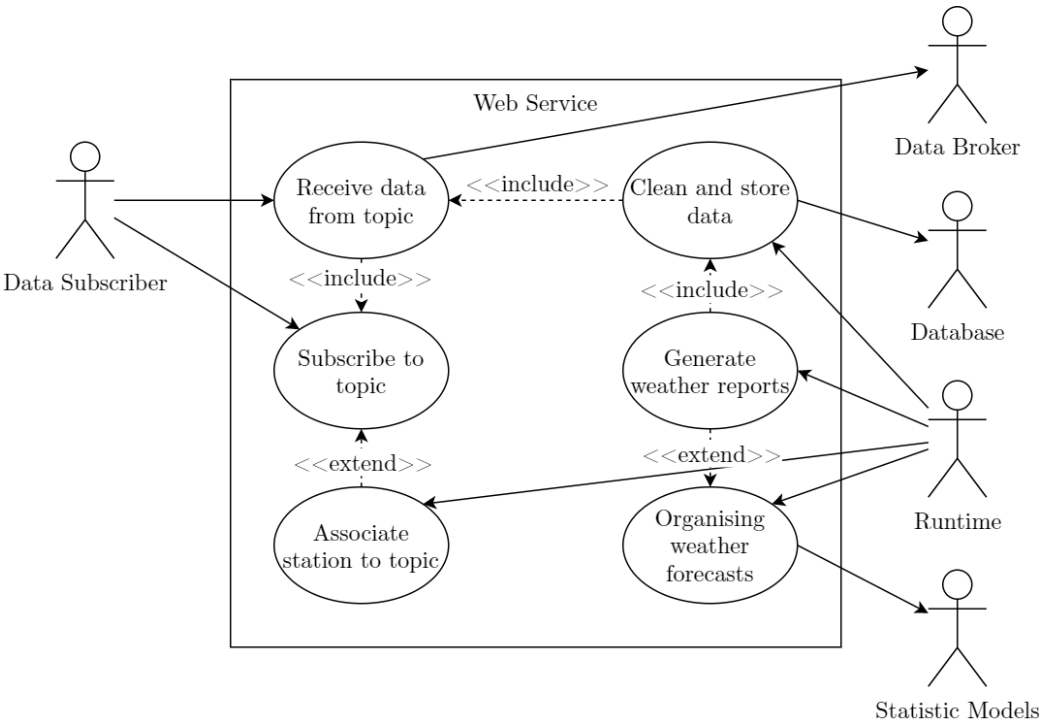

Figura 3.3: Caso de utilização - Processamento de dados

Finalmente, o caso de utilização apresentado na Figura 3.4 o ponto de vista do utilizador final para a interface *Web*, seja este um administrador, dono de estação ou visitante. Como tal, o utilizador pode ter três níveis de permissão, sendo que cada nível superior acrescenta novas funções às funções do seu inferior.

No nível mais baixo, como visitante, é potenciado ao utilizador a visualização de dados de qualquer estação, tanto em forma de boletim como em gráficos históricos. Para além disso, é disponibilizado ao utilizador um serviço de notificações de eventos relacionados ao dados das estações.

No nível intermédio, como participante no sistema com as suas próprias estações meteorológicas, é potenciado ao utilizador o registo de uma conta. Através da autenticação na mesma este pode registar as estações que possui e associar às mesmas atributos, como a sua localização.

Finalmente, o nível máximo de permissão disponibiliza as funcionalidades de aceder a registos do sistema, gerir utilizadores e estações.

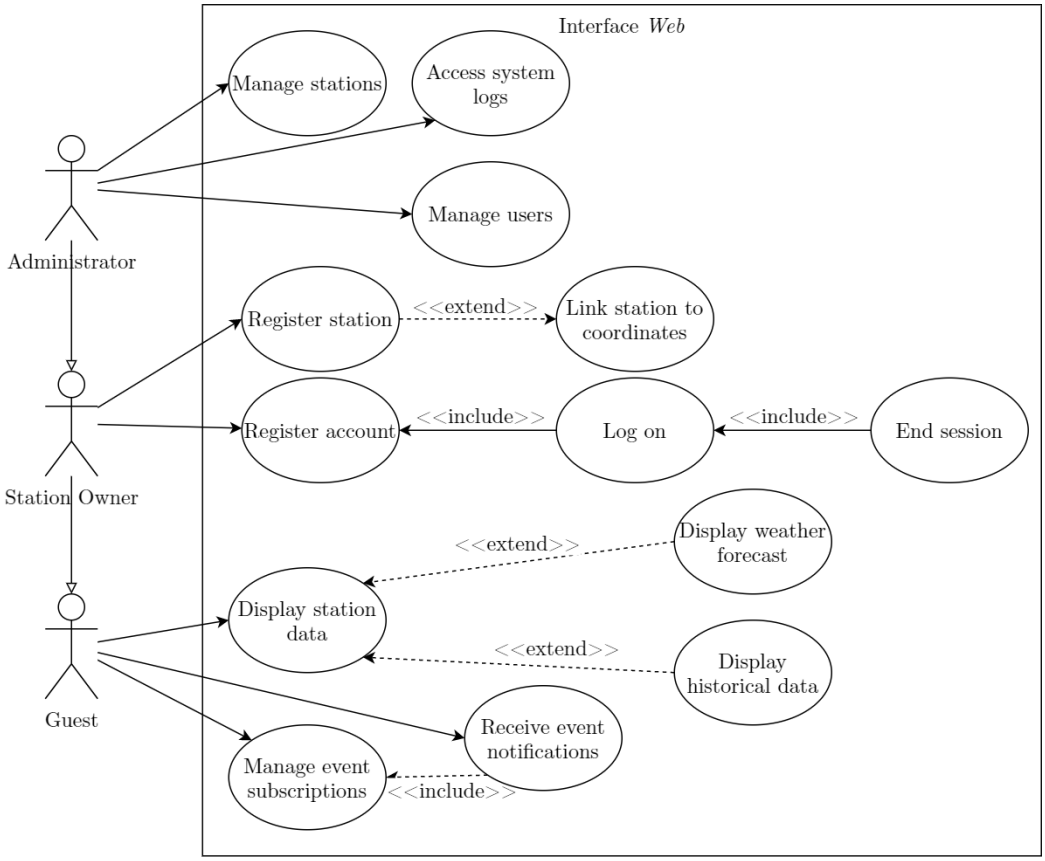

Figura 3.4: Caso de utilização - Exposição dos dados

3.3.1 Escalonamento

Com base nas métricas descritas na Tabela 3.3, a prioridade dos casos de utilização foi calculada, resultando na Tabela 3.4.

Tabela 3.3: Métricas para escalonamento dos casos de utilização

a	Envolve esforço de investigação em tecnologias novas ou arriscadas
b	Inclui funções de alta complexidade ou de tempo de resposta crítico
c	Tem alto contributo na adição das classes que descrevem os conceitos do domínio, ou requer serviços especiais de suporte à persistência
d	Exige grandes volumes de informação ou detalhado conhecimento de negócio
e	Representa processo essencial na linha de negócio
f	Suporta diretamente o retorno do investimento ou tem alta contribuição na redução de custos

Tabela 3.4: Matriz para decisão sobre prioridade dos casos de utilização

Ator	**Caso de utilização**	**Peso**	**a** **3**	**b** **1**	**c** **2**	**d** **1**	**e** **1**	**f** **1**	**Soma**
Destinatário de dados	Receber dados de tópico		2	2	4	3	4	3	26
Runtime do microprocessador	Ler dados para *thread*		2	4	4	2	4	1	25
Runtime do serviço	Organizar uma previsão meteorológica		4	4	1	4	1	1	24
Runtime do serviço	Gerar boletins meteorológicos		3	1	2	4	2	1	21
Runtime do microprocessador	Recolher resultados com marca horária		2	2	2	3	4	2	21
Visitante	Consultar boletim meteorológico		3	2	1	2	3	1	19
Remetente de dados	Publicar dados em tópicos		1	1	2	3	4	3	18
Destinatário de dados	Subscrever a tópico		1	1	4	1	4	0	17
Visitante	Receber notificações de eventos		3	4	0	2	2	0	17
Dono de estação	Registar estação		1	2	3	1	4	0	16
Visitante	Consultar histórico de dados		2	1	1	4	1	2	16
Visitante	Gerir inscrição em eventos		3	2	0	3	2	0	16
Administrador	Consultar logs do sistema		2	2	0	2	0	3	13
Runtime do serviço	Associar estação a tópico		1	0	3	1	3	0	13
Runtime do serviço	Limpar e armazenar dados		2	2	0	4	0	0	12
Administrador	Gerir estações		1	1	2	1	2	0	11
Dono de estação	Registar-se		1	1	2	1	2	0	11
Administrador	Gerir registo de utilizadores		1	1	0	3	2	0	9

A partir dos resultados apresentados na última coluna da Tabela 3.4 conclui-se que o sistema é sensível à persistência entre o lado do remetente e destinatário de dados, uma vez que poderão ser perdidas mensagens caso algum dos lados não esteja pronto para as receber. Igualmente preocupante no desenvolvimento será o acoplamento dos sensores com o microprocessador, dado que este passo serve como base para o funcionamento de todo o restante projeto desenvolvido. No lado inverso, nota-se que não é tão prioritária a implementação das funções do administrador, uma vez que estas apenas servem para resolver problemas que os utilizadores não consigam resolver por si mesmos.

3.4 Modelo de dados

O modelo Entidade-Associação (EA) é um modelo de dados abstracto, que define uma estrutura de dados ou de informação que pode ser implementada numa base de dados relacional. É composto por entidades, que classificam objetos de interesse, e especifica as relações entre estas.

Num modelo EA que esteja descrito utilizando a notação Chen [12], as entidades são representadas como retângulos, os atributos como ovais e as relações como losangos.

As ligações podem ser uma linha ou pé-de-galo, representando uma ou muitas respetivamente, acompanhadas de um círculo ou linha, representando opcional ou obrigatória. Na conversão para base de dados relacional, é regra geral que cada relação 1:1 é resumida por uma tabela, 1:N por duas e M:N por três tabelas.

As formas contornadas por duas linhas são classificadas como fracas (entidades ou relações) e representam o caso em que essa entidade é dependente da referência a outra para se identificar (a chave estrangeira faz parte da sua chave primária).

Os atributos com texto sublinhado denotam um candidato a chave primária, enquanto que os contornados a tracejado são derivados a partir de outros. Uma ligação a tracejado de uma entidade ao atributo denota a opcionalidade do mesmo.

Em concreto, a Figura 3.5 apresenta o modelo EA simplificado que foi desenvolvido para suportar a base de dados utilizada neste projeto. Neste modelo é possível ver dois ramos associados ao utilizador.

O primeiro ramo associa este aos eventos (`Event`) registados a partir de uma subscrição (`Subscription`), enquanto que o segundo ramo associa cada conta de utilizador (`User`) a zero ou mais estações (`Station`).

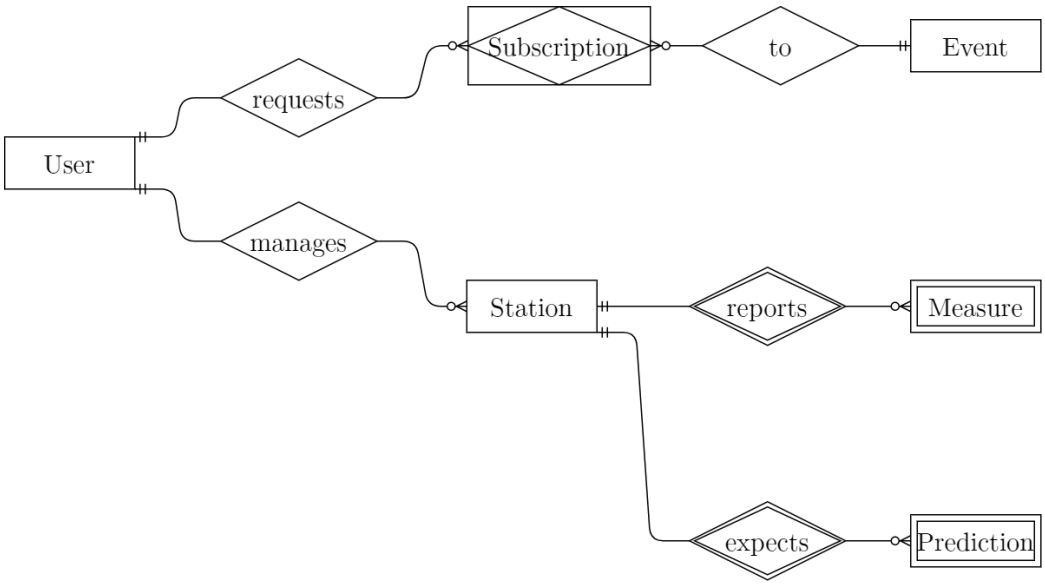

Figura 3.5: Modelo simplificado Entidade-Associação da base de dados, sem atributos visíveis

Os utilizadores (entidade `User` da Figura 3.6) têm como atributos o seu endereço de correio eletrónico, uma versão cifrada da sua palavra-passe, o seu nível de permissão e finalmente um atributo booleano que indica se a conta foi ativada ou não através de *e-mail*.

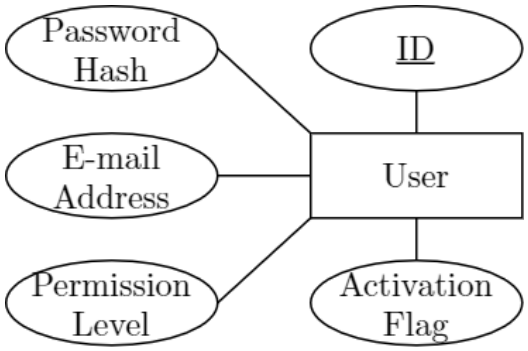

Figura 3.6: Entidade "Utilizador" do modelo de dados e os seus atributos

Estação (entidade `Station` da Figura 3.7) é uma entidade simples que contém como atributos o nome do tópico que a identifica, o seu nome completo do qual é derivado o tópico ao ser criada, a sua localização em coordenadas geográficas e uma descrição opcional.

Os elementos Estação fazem parte da chave primária das medidas e previsões, que são entidades semelhantes, às quais são atribuídos objetos JSON que contém variadas variáveis meteorológicas. Cada medição, para além da estação envolvida, é indexada pela sua marca temporal.

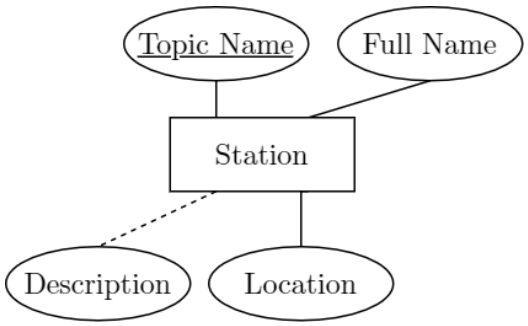

Figura 3.7: Entidade "Estação" do modelo de dados e os seus atributos

As entidades Medida e Previsão (apresentadas juntas como `Measure` e `Prediction` respetivamente na Figura 3.8) partilham os mesmos atributos e

devem ser compatíveis entre elas. O objetivo da separação das mesmas é a distinção entre os dados que são reais e os gerados pelo mecanismo de previsões. Estas entidades devem incluir todo o tipo de variáveis meteorológicas, desde as básicas como temperatura e humidade às calculadas com base nas outras como é o caso do ponto de condensação da água.

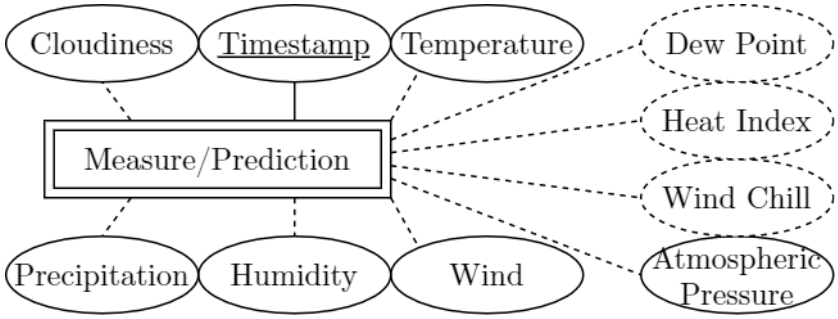

Figura 3.8: Entidades "Medida" e "Previsão" do modelo de dados e os seus atributos condensadas numa só

Os eventos (entidade `Event` na Figura 3.9) são definidos pelas três componentes da equação que os representa. A primeira é a variável a testar, que será mapeada num dos atributos das medições mencionadas no parágrafo anterior. De seguida, é escolhido o sinal de inequação que se procura na relação entre a variável e a última, o valor, que é o limiar sobre o qual é ativado o evento.

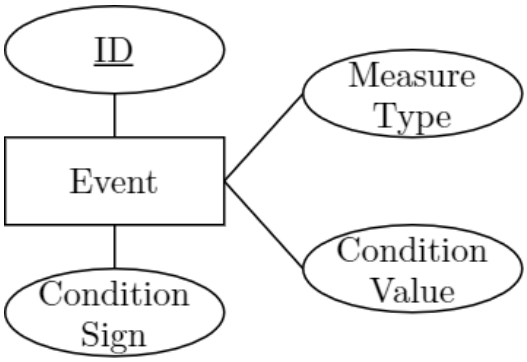

Figura 3.9: Entidade "Evento" do modelo de dados e os seus atributos

3.5 *Mockups* da interface gráfica

A presente secção apresenta o planeamento relativamente ao aspeto das páginas *Web* que serão desenvolvidas, tradicionalmente chamado de *mockup*.

As hiperligações nas páginas apresentadas nas Figuras 3.10 a 3.18 são representadas por texto sublinhado ou como botões (retângulos arredondados). Os balões de fala apresentam alternativas à apresentação do elemento para onde apontam, por exemplo um menu que só é vísivel enquanto o utilizador tem sessão iniciada.

O texto a cinzento representa espaços editáveis pelo utilizador, ou seja, elementos HTML *input*, as entradas dos formulários.

Todas as páginas são constituídas por um cabeçalho, um rodapé e um corpo. O cabeçalho têm duas versões, retratadas na Figura 3.10. Uma das versões para um utilizador com sessão iniciada, com uma mensagem de boas vindas, botão de *logout* e um menu com botões para as outras páginas. A outra versão, para utilizadores sem sessão, tem apenas um botão de *login*. Ambas as versões apresentam à esquerda um título, ao centro as coordenadas obtidas pelo serviço anteriormente mencionado e finalmente à direita uma barra de pesquisa.

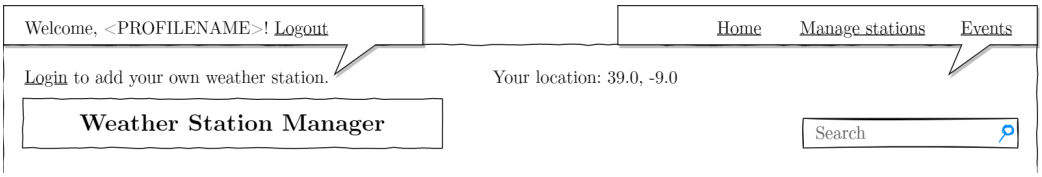

Figura 3.10: *Mockup* do cabeçalho

O rodapé, apresentado na Figura 3.11 contém informação geral acerca do projeto e ocupa sempre uma porção do fundo da página.

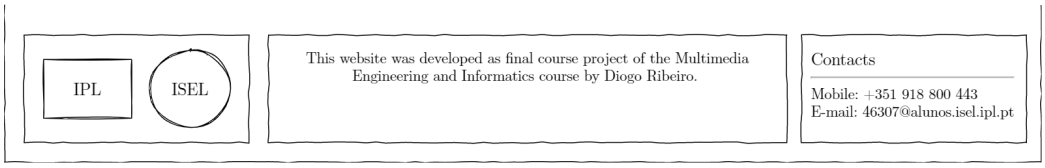

Figura 3.11: *Mockup* do rodapé

Dado que o cabeçalho e o rodapé de todas as páginas é igual (dependendo se o utilizador está ou não autenticado) nos seguintes *mockups* apenas se apresenta o corpo das páginas.

A página inicial do serviço na Figura 3.12 apresenta ao utilizador uma lista de todas as estações registadas no mesmo, ordenadas pela distância ao utilizador. Esta distância é obtida a partir de um serviço de geolocalização pelo endereço IP.

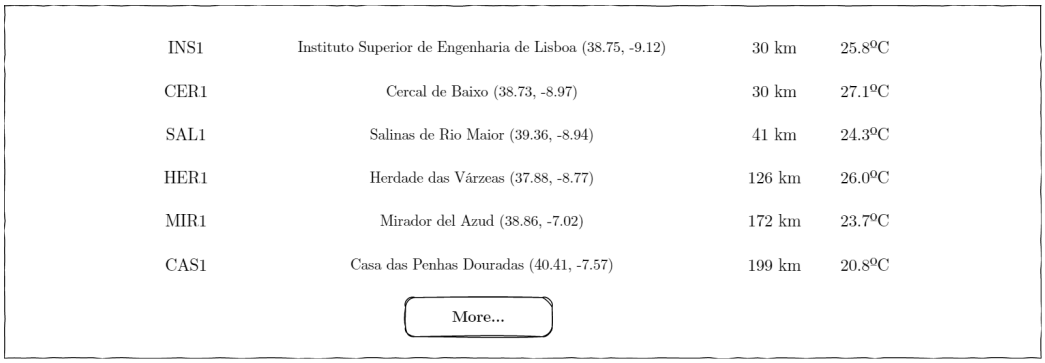

Figura 3.12: *Mockup* da página inicial

As páginas de *login* e registo na Figura 3.13 interligam-se e consistem em formulários que correspondem à norma habitual, sendo que têm botões dedicados a trocar a visibilidade da palavra-passe e pedir para recuperar a palavra-passe.

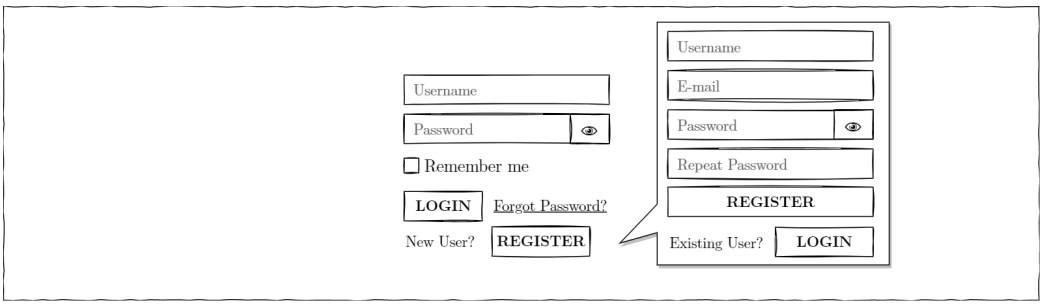

Figura 3.13: *Mockup* das páginas de *login* e registo

Um utilizador com sessão iniciada pode aceder, a partir do menu no cabeçalho, à página da Figura 3.14 onde consta a lista de todas as suas estações com botões para edição, adição ou remoção destas.

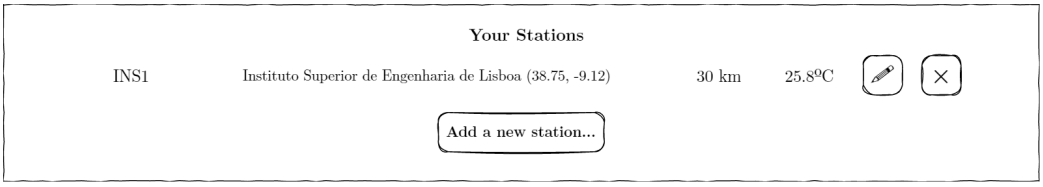

Figura 3.14: *Mockup* da página da listagem de estações do utilizador

Ao premir um dos botões previamente mencionados, o utilizador acede à página de edição da estação, que difere da página de adição apenas no botão de submissão do formulário. Estas páginas, representadas na Figura 3.15 consistem numa tabela com os atributos da estação, que podem ser introduzidas pelo utilizador.

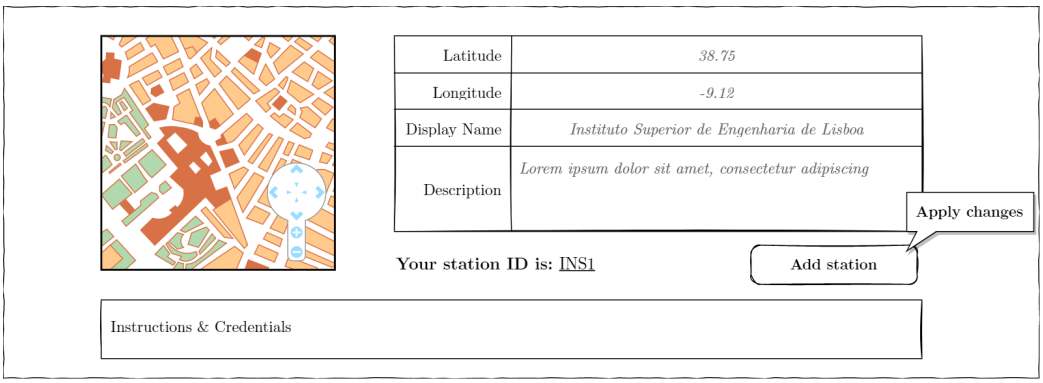

Figura 3.15: *Mockup* da página de adição ou edição de estações

Ao clicar no botão de eventos no menu é apresentado ao utilizador uma página com uma tabela onde lhe é potenciada a criação ou cancelamento de eventos. Esta está apresentada graficamente em maior detalhe na Figura 3.16.

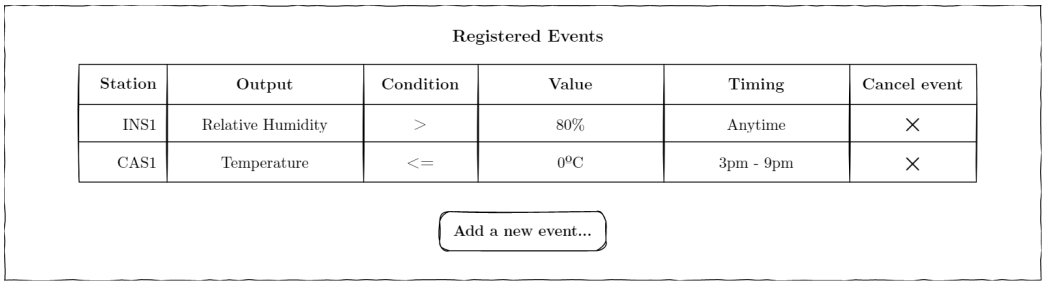

Figura 3.16: *Mockup* da página de gestão de eventos

A Figura 3.17 demonstra que para adicionar um evento é disponibilizado ao utilizador um formulário onde este pode escolher a estação e designar a condição pela qual quer ser alertado através do e-mail registado.

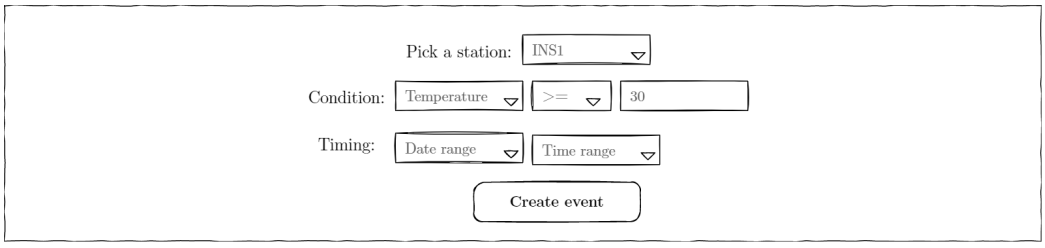

Figura 3.17: *Mockup* da página de criação de eventos

Resumindo, as páginas contém hiperligações que resultam nas interligações apresentadas na Figura 3.18.

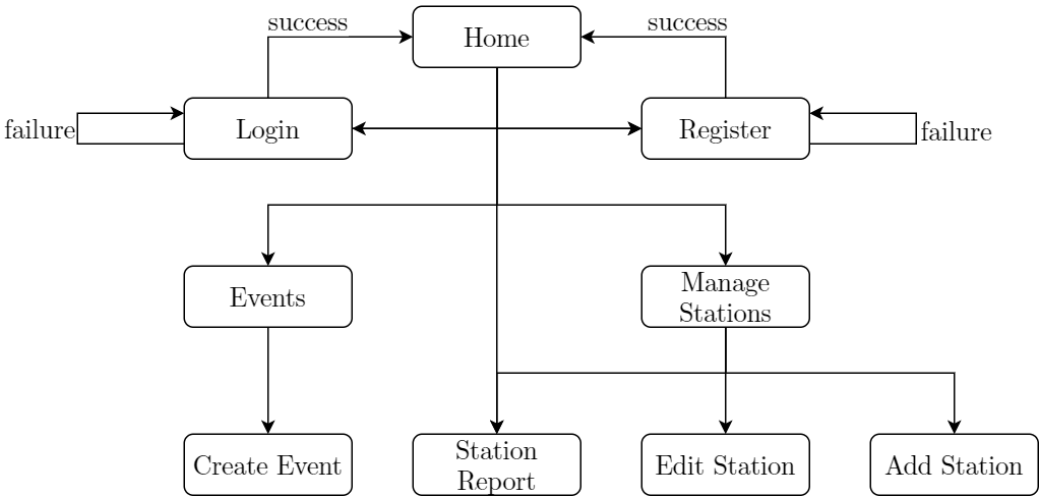

Figura 3.18: Diagrama de navegação pelo hipertexto da interface Web

Capítulo 4

Implementação do Modelo

Este capítulo descreve a arquitetura do sistema proposto na Secção 4.1. De seguida as Secções 4.2 a 4.9 detalham para cada uma das restantes partes as especificações das tecnologias utilizadas, as motivações para a sua seleção e o processo de construção do sistema.

4.1 Arquitetura proposta

A partir da arquitetura consideram-se duas partes: uma focada na individualidade da estação meteorológica prototipada e outra no serviço disponibilizado às múltiplas estações que se inscreverem.

As Figuras 4.1 e 4.2 apresentam as partes mencionadas no parágrafo anterior num nível de detalhe superior à Figura 3.1.

Na primeira das figuras foi tida em consideração a estação não comercial em desenvolvimento. Prevê-se que a partir dessa estação vão ser medidas as variáveis meteorológicas. Incorporado nesta, um sensor BME280 [13] mede temperatura, humidade e pressão. Este sensor comunica por I2C com o microcontrolador. Às variáveis meteorológicas mencionadas anteriormente acrescenta-se o vento: a sua velocidade é medida a partir de um anemómetro e a sua direção a partir de um catavento. Finalmente, é medido o volume de precipitação a partir de um pluviómetro. Estes dados são recolhidos e enviados para um tópico no *broker* MQTT.

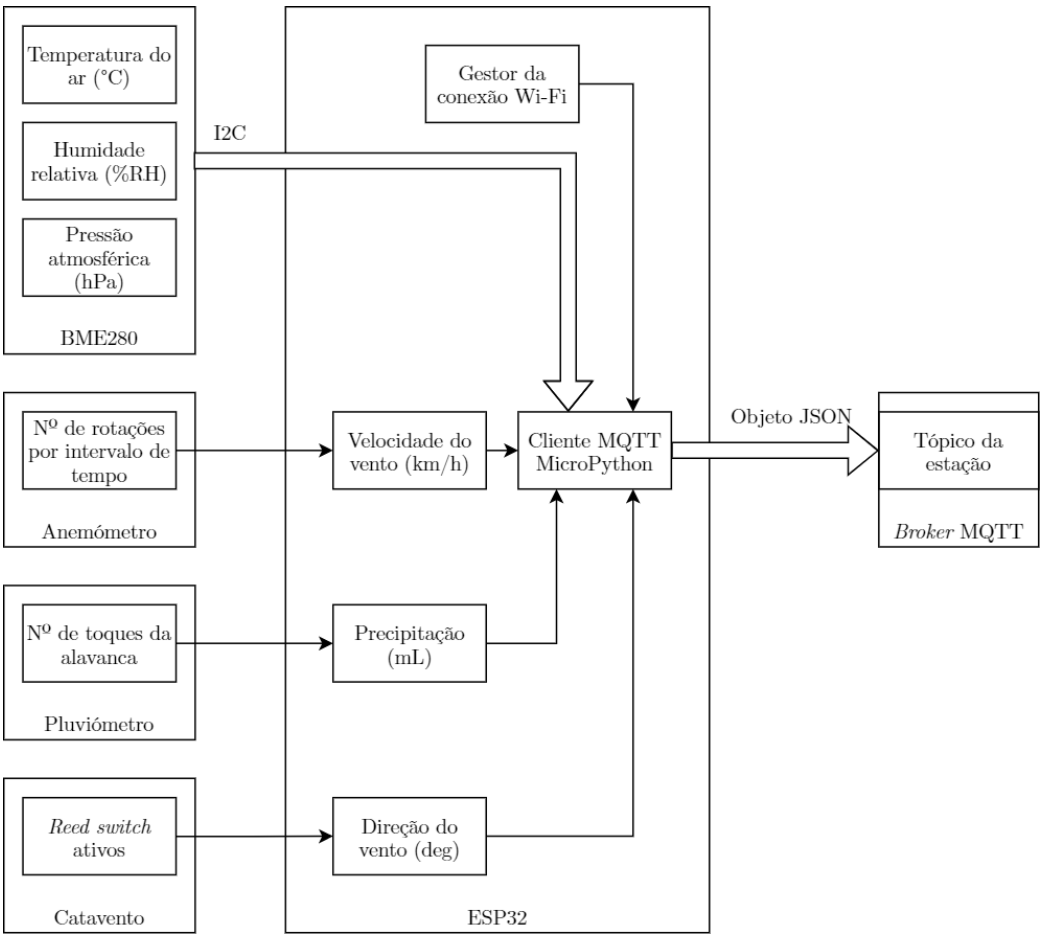

Figura 4.1: Diagrama de blocos para a estação

O serviço MQTT complementa o diagrama anterior ao partilhar o *broker* com várias estações, recolhendo com um cliente MQTT a informação que estas publicam nos seus tópicos. Esta informação, depois de processada, é armazenada na base de dados. Este processamento é detalhado na Secção 4.7.

A partir desta informação são realizadas as previsões e construídos os gráficos para as páginas *Web*. Finalmente, o sistema interage com o utilizador final através dos protocolos HTTP para as páginas *Web* e SMTP para as notificações de correio eletrónico.

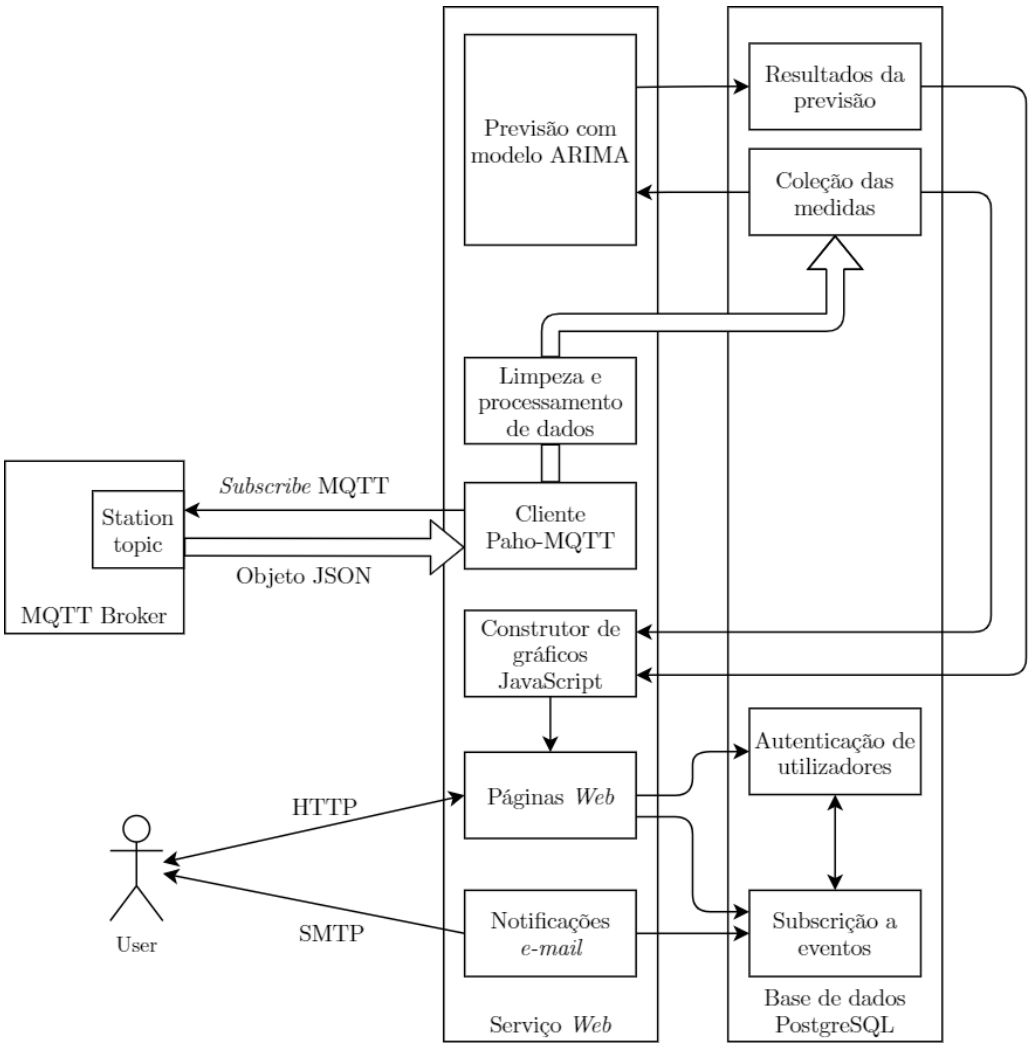

Figura 4.2: Diagrama de blocos para o serviço meteorológico

4.2 Microcontrolador

O microcontrolador ESP32 é um sistema de baixo custo e baixa potência numa *Series of Chips* (SoC) com capacidades Wi-Fi e *Bluetooth dual-mode*.

É importante ter em consideração que um ESP32 tem três níveis hierárquicos: a placa de desenvolvimento contém um módulo que por si contém um microcontrolador.

Comparando os diferentes modelos de ESP32, existem versões ligeiramente diferentes. A placa de desenvolvimento utilizada no projeto em estudo

é uma não oficial baseada num módulo ESPRESSIF-ESP32-WROVER-B e como tal tem um microcontrolador ESP32-D0WD. Este microcontrolador trabalha com tensão de 2.3 a 3.6 Volt e corrente até cerca de 240 miliamperes, pico atingido ao transmitir tramas Wi-Fi. A sua arquitetura é apresentada na Figura 4.3:

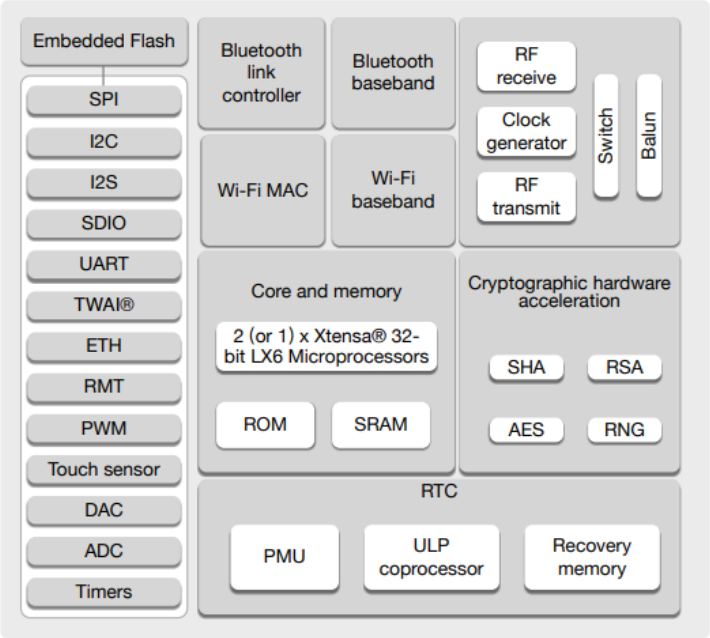

Figura 4.3: Arquitetura do microcontrolador ESP32
Fonte: [14]

Os módulos ESP32 são incluidos nas placas de desenvolvimento e variam no microcontrolador que contém, dimensões, número de pins, possíveis tamanhos de memória tanto flash como PSRAM e antena (habitualmente PCB ou conector IPEX).

Por outro lado, os microcontroladores ESP32 diferenciam-se pelo número de *cores* (sendo que maior parte da geração atual têm 2), pinos, RAM e tamanho das memórias flash e PSRAM.

O microcontrolador ESP32, com base na sua relação qualidade/preço, pequena dimensão e consumo de energia relativamente baixo, está bem adaptado a uma série de diferentes aplicações IoT.

4.2.1 MicroPython

MicroPython [15] é uma implementação simplificada da linguagem de programação Python 3 que inclui um subconjunto de bibliotecas *built-in* Python e está otimizada para funcionar com microcontroladores. Uma vantagem do Python como linguagem de programação é que pode ser aprendido rapidamente, e em maior parte dos casos o código transmite de forma clara a intenção do programador.

Para instalar o *firmware* MicroPython no microcontrolador, começou-se por ligar a placa de desenvolvimento à corrente através de um adaptador Micro USB 2.0 entre a placa e o computador. Depois de extrair para o sistema de ficheiros do computador o binário disponibilizado.

Para carregar o *firmware* para a placa foi necessário primeiro que se ativasse o modo *bootloader*. Isto fez-se premindo o botão no canto da placa com a etiqueta "BOOT" ao iniciar. De seguida, foi instalado através da ferramenta *pip* a ferramenta *esptool* que permite através da linha de comandos limpar a memória *flash* e instalar o binário na placa através dos comandos representados no Código 4.1. Note-se que foi necessário também alterar o parâmetro BAUD da comunicação série para 115200 neste processo.

Código 4.1: Comando para dar *flash* ao firmware

```
1 C:\> esptool.py --port COM3 erase_flash
2 C:\> esptool.py --chip esp32 --port COM3 write_flash -z 0x1000
  ↪  esp32-idf4-20210202-v1.14.bin
```

Completada a instalação, pode ser verificado através do uso da REPL (consola de Python interativo) que versão instalada é a 1.14.0 [15], que segue a implementação de Python 3.4. O resultado desse teste está representado na Figura 4.4.

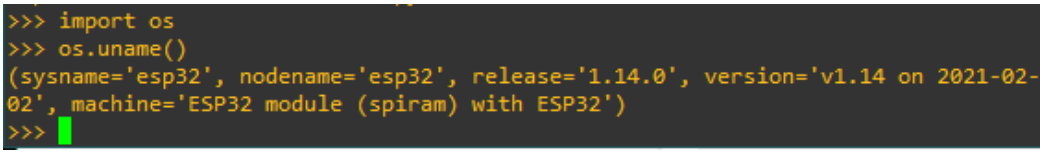

Figura 4.4: Verificação da versão de MicroPython no REPL através de PuTTY

Se precisarmos de utilizar funções de uma biblioteca que não esteja implementada no *firmware* fundamental do MicroPython, esta biblioteca deve ser transferida sob a forma de um módulo Python para a memória persistente do dispositivo. Em maior parte dos casos, estas são bibliotecas que fornecem comunicação e comportamento de um determinado elemento de hardware, sensor ou atuador. O carregamento do programa e das funções da biblioteca para o microcontrolador foi feito utilizando a ferramenta Python *ampy*, também instalada por *pip*, que fornece comunicação com o microcontrolador através da porta série. O comando para carregar um ficheiro `script.py` para o microcontrolador assemelha-se ao apresentado no Código 4.2:

Código 4.2: Exemplo de comando para transferir ficheiro para ESP32 por porta série

```
1 C:\> ampy --port COM3 put script.py
```

Uma ligação terminal pode ser utilizada para *debug* a Python num microcontrolador, por exemplo, utilizando o programa PuTTY [16]. Pode-se usar qualquer editor de texto disponível para escrever programas Python. Foi utilizado o Visual Studio Code [17] para este fim. Esta abordagem de desenvolvimento é vantajosa para a prototipagem e desenvolvimento de novas soluções algorítmicas para sistemas embebidos baseados em *firmware* MicroPython.

4.3 Montagem

A placa de desenvolvimento foi acoplada a uma *breadboard* onde foram instalados em paralelo os sensores BME280 (temperatura, humidade e pressão),

VEML6075 (sensor de luz ultravioleta) e TSL2561 (sensor de luz visível e infravermelhos). Os últimos dois sensores mencionados não foram utilizados mas potenciam trabalho futuro caso se tencione adicionar novas variáveis meteorológicas ao sistema, por exemplo o índice ultravioleta. Para além destes, foi instalado um interruptor simples a um *pull-up* interno. O botão manual substitui como solução temporária o sinal do anemómetro, que completa o circuito duas vezes por cada volta completa (360 graus) que as pás completem.

O circuito foi soldado numa placa de prototipagem, permitindo a montagem num circuito menos frágil, evitando danos aos fios ao utilizar a funcionalidade portátil da placa. A Figura 4.5 apresenta uma fotografia da montagem final.

Figura 4.5: Montagem do circuito na placa de prototipagem

4.4 Comunicação entre estação e servidor

Para a transferência de dados entre o microcontrolador e o servidor de comunicação optou-se pelo formato JSON. Comparado com as alternativas YAML

e XML, JSON é o formato com melhor suporte num contexto *Web*, dado que pode ser importado diretamente como objeto JavaScript, e é melhor incorporado pelas bases de dados.

MQTT é um protocolo de rede simples e flexível. Especializado para aplicações IoT, a sua simplicidade permite a sua implementação tanto em dispositivos com hardware limitado como em redes de alta latência ou largura de banda limitada.

O canal selecionado para o efeito foi um canal MQTT, sendo que a sua alternativa seria um canal HTTP (RESTful). Comparando ambos, MQTT seria mais indicado para poupar energia e largura de banda através da troca de mensagens simples, das quais apenas a última estaria disponível, enquanto que APIs RESTful suportam transferências de grandes documentos e pedidos mais específicos, por exemplo para um intervalo temporal. O *broker* utilizado para os testes MQTT foi disponibilizado pelo orientador.

O canal através do qual as mensagens JSON são enviadas pode ser, opcionalmente, cifrado por SSL/TLS, configurável a partir do *broker*. No entanto, considerando-se que os dados em trânsito não são sensíveis, foi colocada pouca prioridade na segurança dos mesmos. O formato JSON permite, em contraste a enviar cada medição individualmente, que os dados sejam agregados juntamente com uma marca de tempo. Pelo efeito, reduz-se a sobrecarga associada aos cabeçalhos das mensagens transmitidas.

4.5 Software da estação

O software desenvolvido para o microcontrolador está separado ao longo de seis módulos. Foram adaptadas da Internet as bibliotecas de suporte à comunicação I2C, ao sensor BME280 (que utiliza a anterior) [18] e à comunicação MQTT [19]. Foi desenvolvido código personalizado dedicado à configuração da estação, o seu ciclo de funcionamento e ainda a biblioteca para medir os valores do anemómetro.

O diagrama na Figura 4.6 apresenta um diagrama de classes para o software desenvolvido para o microcontrolador. Centrando-se na classe Estação (classe `Station`), a figura apresenta a relação de herança com o cliente MQTT (classe `MQTTClient`), dado que acrescenta às suas funcionalidades a ligação WLAN e a construção das mensagens. Para além disso, mostra a dependência

que a estação tem das classes responsáveis pela leitura dos sensores, sendo que no caso dos sensores que comuniquem por I2C estes são generalizados numa classe abstrata `Device`.

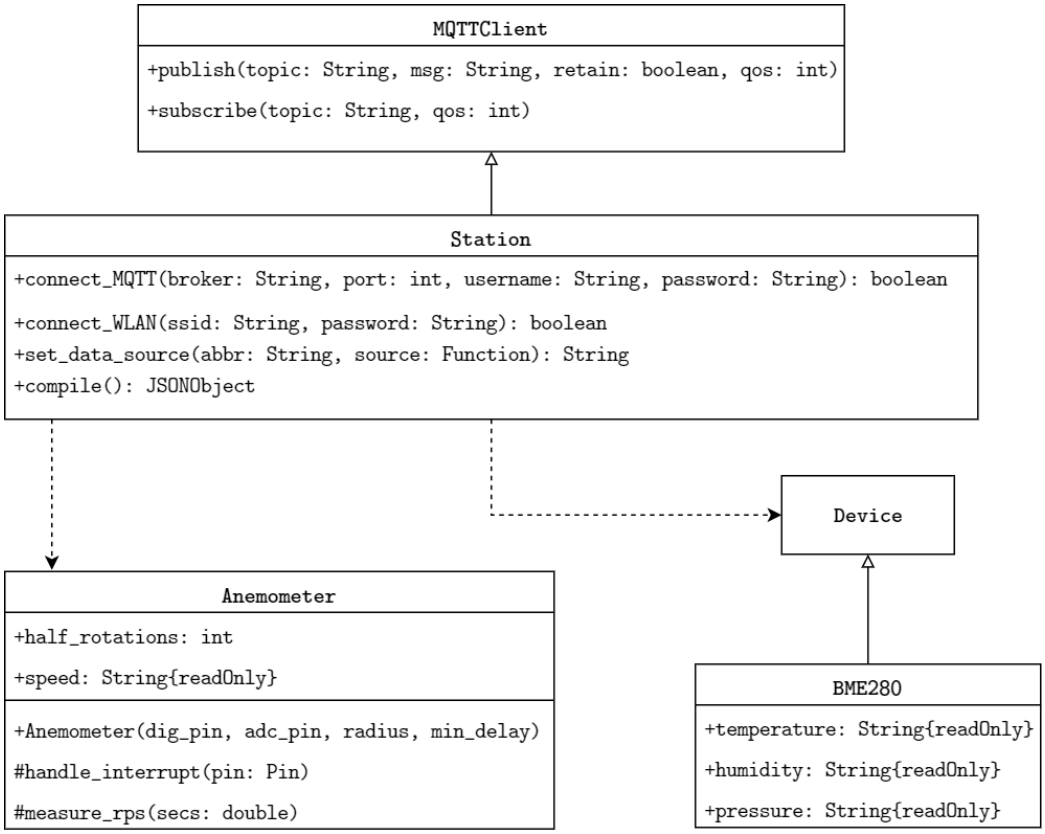

Figura 4.6: Diagrama UML de classes simplificado do software MicroPython desenvolvido

4.5.1 Módulo I2C e BME280

O módulo I2C consiste numa classe `Device`. Esta classe tem métodos para ler e escrever em formato de *bytes* em memória, utilizando o endereço I2C do sensor.

O sensor BME280 está configurado numa classe que faz uso da classe `Device` com uma quantidade de constantes que descrevem o endereçamento dos registos no sensor. Para além disso, têm três métodos públicos, apresentados na Listagem 4.3 que servem como saídas dos dados com dois decimais de precisão para valor JSON.

Código 4.3: Funções de saída para a classe `BME280`

```
def temperature(self):
    "Return the temperature in degrees."
    t = self.read_temperature()
    ti = t // 100
    td = t - ti * 100
    return "{}.{:02d}".format(ti, td) # C

def pressure(self):
    "Return the temperature in hPa."
    p = self.read_pressure() // 256
    pi = p // 100
    pd = p - pi * 100
    return "{}.{:02d}".format(pi, pd) # hPa

def humidity(self):
    "Return the humidity in percent."
    h = self.read_humidity()
    hi = h // 1024
    hd = h * 100 // 1024 - hi * 100
    return "{}.{:02d}".format(hi, hd) # %
```

4.5.2 Módulo MQTT

Seguindo para o módulo MQTT, este define uma classe `MQTTClient` com maior parte das funcionalidades permitidas pela versão 5.0. O construtor desta classe armazena como atributos o endereço, porto, *username* e *password* do *broker*, sendo que ao ser chamado o seu método `connect` cria uma ligação por *socket* ao *broker*, construíndo a mensagem manualmente e enviando para a *socket*. Para além deste, a classe tem também métodos para subscrever a tópicos e publicar mensagens nestes. Estes tópicos recebem como argumentos o *payload*, o nome do tópico e parâmetros QoS e *retain* no caso de publicar, para além de terem em conta outras *flags* definidas no construtor.

O parâmetro QoS é representado por valores de zero (0) a dois (2). O valor zero (0) significa na prática que a mensagem vai chegar no máximo uma

vez ao seu destinatário, ficando implícito que pode também não chegar. A esta estratégia costuma-se dar o nome de "*best-effort*". O valor um (1) indica que a mensagem vai chegar pelo menos uma vez ao destinatário, sendo que para tal é necessário que o remetente a reenvie as vezes necessárias em caso de erro. Finalmente, o valor dois (2) indica que é estabelecida uma ligação entre remetente e destinatário de forma à mensagem chegar ao outro lado exatamente uma vez. O valor que melhor mistura custo e consistência é `QOS=1`, pelo que o parâmetro utilizou esse valor ao longo do projeto desenvolvido.

A *flag* `retain` ativa indica que a mensagem será armazenada no *broker* para, caso o destinatário não esteja subscrito ao tópico no momento do envio, possibilitar a sua receção no futuro. Dependendo do QoS, podem ser retidas todas as mensagens ou apenas a última num tópico.

4.5.3 Módulo do anemómetro

Para ler o anemómetro foi considerado que o mecanismo ativaria um interruptor duas vezes por cada volta que completasse. Desta forma, para calcular a velocidade do vento, é preciso ter em conta o raio ou diâmetro desse mecanismo. Para além disso, é preciso adicionar um período de *debounce* para que cada subida de tensão durante a ativação do interruptor não resulte em várias chamadas ao *interrupt* configurado. O bloco de código 4.4 apresenta o método da classe configurado para execução no evento de subida do sinal digital de 0 para 1. Note-se que a variável `min_delay` está expressa em nanossegundos e a variável `radius` em metros, ou seja, é esperado que o intervalo entre meias-voltas seja pelo menos 20 milissegundos, resultando numa velocidade máxima do vento de $2\pi * f * r = 2\pi * \frac{1}{0.2*2} * r = 2\pi * 25 * 0.3 \approx 47.12\, m/s$ para um anemómetro com 30 centímetros de raio, onde f é a frequência ou número de rotações por segundo e r o raio do círculo.

O valor escolhido para o intervalo, 20 milissegundos, foi determinado durante os testes com o botão ter folga suficiente para evitar falsas ativações consecutivas. O valor do raio foi medido a partir do protótipo do anemómetro preparado.

A maior velocidade de vento alguma vez registada foi 113.3 metros por segundo em Barrow Island na Austrália. Para casos extremos como este pode ser diminuído o atributo `min_delay` até ao limite em que o intervalo mínimo não é suficiente para diferenciar entre duas meias-voltas diferentes ou ainda

aumentado o raio do anemómetro.

Código 4.4: Função de *interrupt* para contar rotações do anemómetro

```
def __init__(self, dig_pin, adc_pin=None, radius=.3, min_delay=20000000):
    (...)
    self._dig_pin.irq(trigger=Pin.IRQ_RISING,
    ↪  handler=self._handle_interrupt)

def _handle_interrupt(self, pin):
    aux = time_ns()
    dtime = aux - self._last_event_time
    if dtime > self._min_delay:
        self.half_rotations += 1
        self._last_event_time = aux
```

Para o cálculo de uma velocidade linear do vento é utilizado o valor de rotações por segundo com a fórmula da velocidade angular e depois convertida em velocidade linear sobre o perímetro do círculo ($2\pi r$).

A aplicação faz uma pausa bloqueante durante um intervalo configurável de 10 segundos para escutar o número de meias-rotações desse mesmo período. Esse valor, quando dividido pelo intervalo de tempo esperado, resulta no número de rotações por segundo. As rotações por segundo multiplicadas são uma frequência que pode ser multiplicada por 2π, o ângulo em radianos para uma volta completa, para obter uma velocidade angular. Esta velocidade angular, quando multiplicada pelo raio, é convertida de radianos por segundo para metros por segundo. O raciocínio utilizado para estes cálculos baseia-se na fórmula 4.1:

$$v = \omega r \qquad \omega = \frac{\Delta\theta}{\Delta t}$$

Equação 4.1: Relação entre velocidade angular (ω) e linear (v)

$\Delta\theta$ representa o ângulo percorrido durante o intervalo de tempo Δt

A última parte dos cálculos, dedicada à construção do valor para envio,

é feita nos métodos apresentados no Código 4.5. O resultado é retornado em formato decimal em quilómetros por hora.

Código 4.5: Função de *interrupt* para contar rotações do anemómetro

```
def _measure_rps(self, secs=10):
    rot_start = self.half_rotations
    start = time_ns()
    c = start
    while True:
        if time_ns() - start > secs*1e9:
            rot_end = self.half_rotations
            r = (rot_end - rot_start)/2
            rps = r / secs
            print("Registered "+str(r)+" rotations in "+str(secs)+" secs
            ↪  ("+str(rps)+" rps).")
            return rps

def speed(self):
    rps = self._measure_rps()
    # dtime = 1./rps if rps else 0
    w = 2*math.pi * rps # angular velocity
    v = w * self._radius # linear velocity

    # Convert to kmh
    s = v * 3.6
    return s
```

Foi feita uma tentativa para calcular a direção também, que dependeria do mapeamento entre os possíveis níveis da corrente e os *reed switch* ativos por ação de um ímane.

4.5.4 Módulo da estação

O módulo Python `station` implementa uma classe `Station` que implementa o estabelecimento da ligação WLAN e MQTT, sincronização NTP, compilação e publicação dos dados obtidos pelos sensores no *broker*.

A sincronização NTP mencionada serve para o método `compile`, apresen-

tado no Código 4.6. É crítico ao JSON que leve uma representação da data e hora atuais de forma a situar as medições no tempo independentemente da altura em que estas são distribuídas pelo *broker*, nomeadamente dado que se pretende ter o parâmetro `retain` ativo e QoS a 1.

Código 4.6: Método que junta as medições num formato JSON para envio

```
def compile(self):
    '''Returns a timestamped dictionary with the latest readings.\n
       Refer to "settings.json" for the keys' names.'''
    res = {key:str(fun()) for key,fun in self._data.items() if fun is
    ↪  not None}
    res["timestamp"] = self.format_time(time.localtime())
    return ujson.dumps(res)
```

O módulo `main` inicializa uma instância da classe `Station` com parâmetros obtidos através da análise de um documento JSON. Inicializa também um objeto para cada sensor, neste caso, `BME280` e `Anemometer`. As funções de saída destes são associadas como *callback* a um dicionário com cada medida através do método da estação `set_data_source`, apresentado no Código 4.7:

Código 4.7: Método que associa funções às chaves JSON das medidas

```
def set_data_source(self, abbr, source):
    '''Stores the sensor's function for future reading
       of whatever measure abbr abbreviates.\n
       Returns the first reading attempt, None if it failed.'''
    self._data[abbr] = source
    if source:
        return source()
```

O método apresentado no Código 4.7 é utilizado no fim da primeira fase de execução do módulo `main`, apresentada no Código 4.8:

Código 4.8: Primeira fase do módulo `main`, equivalente à função `setup` em Arduino

```
with open("settings.json", "r") as f:
    config = ujson.loads(f.read())

mconf = config["mqtt"]
sta = Station(main_topic=mconf["main_topic"])

wconf = config["wlan"]
wlan = sta.connect_WLAN(ssid=wconf["ssid"], password=wconf["password"])

mqtt = sta.connect_MQTT(broker=mconf["broker"], port=mconf["port"],
↪  username=mconf["username"], password=mconf["password"])

bme = BME280(i2c=SoftI2C(scl=Pin(22), sda=Pin(21), freq=10000))
sta.set_data_source('tmp', bme.temperature)
sta.set_data_source('hum', bme.humidity)
sta.set_data_source('atm', bme.pressure)

anemo = Anemometer(dig_pin=14, radius=.3)
sta.set_data_source('ws', anemo.speed)
#sta.set_data_source('wd', anemo.direction)

# pluvio = Pluviometer(i2c=SoftI2C(scl=Pin(?), sda=Pin(?), freq=10000))
# sta.set_data_source('rain', pluvio.precipitation)
```

Depois da fase detalhada no Código 4.8, a estação entra num ciclo entre esperar pelo próximo momento de medição, compilar e publicar os dados, como apresentado no Código 4.9:

Código 4.9: Segunda fase do módulo `main`, equivalente à função `loop` em Arduino

```
while True:
    msg = sta.compile()
    print(msg)
    sta.publish_data(msg)
    time.sleep(60*5-11) # wait 5 minutes minus the ~11 seconds it takes
    ↪  to calculate wind speed
```

4.6 Sistema de base de dados

Para o sistema de base de dados foi selecionado PostgreSQL [20]. Considerou-se uma base de dados não relacional (também chamadas de NoSQL) como a MongoDB. No entanto, ficou decidido que, com base na existência de relações entre utilizadores e eventos, a natureza dos dados deve ser reduzida em tabelas em vez de documentos. Em relação a outras bases de dados relacionais como MySQL, foi tido em consideração que PostgreSQL suporta tipo de dados JSON e geoespaciais com a extensão PostGIS [21], para além de ser possibilitada a herança entre tabelas, e como mencionado anteriormente suporte de tipos de ficheiros como JSON e XML.

A instalação da base de dados em Windows trata-se de um instalador simples a partir do qual o serviço fica disponível sob a aplicação `services.msc` para iniciar ou parar. A versão instalada é a 13.2, evidenciada pelo comando `psql --version`.

4.6.1 Bibliotecas geoespaciais

PostGIS (PostgreSQL Geographical Information System) é uma extensão de base de dados espacial para PostgreSQL. Adiciona suporte para objectos geográficos, permitindo que *queries* baseadas em pontos geográficos sejam executadas em SQL.

GEOS (Geometry Engine - Open Source) é uma adaptação em C++ de um motor Java de funcionalidades geoespaciais. É uma dependência de GDAL.

GDAL é uma biblioteca de tradução de formatos de dados geoespaciais raster e vetoriais lançada pela *Open Source Geospatial Foundation* (OSGeo).

OSGeo4W é uma distribuição binária de um vasto conjunto de software geoespacial de código aberto para ambientes Windows. OSGeo4W inclui aplicações de desktop GIS de código aberto, bibliotecas geoespaciais, linguagens de scripting, bem como muitos outros pacotes relacionados.

PROJ é uma biblioteca destinada à realização de conversões entre projeções cartográficas.

Para instalar as extensões, foi seguido o tutorial disponibilizado pela documentação Django [22]. A partir da aplicação Stack Builder (instalada com a base de dados) foi instalada a versão 3.1.2 de PostGIS. De seguida foi instalado o *bundle* OSGeo4W, que instala diretamente as bibliotecas PROJ, GDAL e GEOS. Tiveram também de ser alteradas algumas variáveis e registos do sistema Windows. A instalação ficou alojada na pasta do disco `C:/`.

4.7 *Framework Web*

Quanto à linguagem de programação para o contexto *Web*, foi preferido Python para facilitar a programação e coesão, para além de ser flexível quanto aos dados recebidos dos sensores.

Outra razão para a escolha do Python é o seu suporte a bibliotecas de análise de dados, especialmente de um ponto de vista polivalente. Para a previsão meteorológica realizada através de modelos matemáticos é facilitado, em relação à linguagem R, a legibilidade e o acoplamento com a base de dados.

A versão de Python utilizada ao longo do projeto desenvolvido é 3.8.5, sendo que foi utilizado um ambiente virtual (`pip install venv`) para que sejam associados diretamente à diretoria do projeto os pacotes necessários.

Das *frameworks* Python mais populares, procurando também boa documentação e contínuo suporte da comunidade, foi selecionado Django [23] em vez de Flask [24] porque que as páginas *Web* planeadas na Secção 3.5 incluem algumas secções onde será mais ergonómico incluir elementos dinâmicos. Flask e Django também se diferenciam pela clareza do código, sendo que Django prefere uma abordagem mais implícita e inclui mais módulos e funcionalidades por omissão que representam convenções do paradigma atual.

Um detalhe da *framework* utilizada é que foram, em conjunção com o sistema de base de dados referido em 4.6, instaladas extensões que permitem a utilização de bibliotecas de software geoespacial *open source*, mencionadas na Subsecção 4.6.1. Estas são incluídas no pacote GeoDjango.

4.7.1 Instalação

Na pasta `django-weather` foi preparado um ambiente virtual Python. Este é ativado ao abrir o projeto correndo o executável `env/Scripts/activate.bat`. As instalações de bibliotecas Python através da ferramenta `pip` são feitas dentro deste contexto, de forma às bibliotecas ficarem alojadas dentro da pasta `env/Lib/site-packages` em vez de junto à instalação original de Python no sistema. Foi gerado um ficheiro `requirements.txt`, disponível na raíz do projeto, de forma a indicar quais foram as bibliotecas incluídas no projeto.

O servidor de desenvolvimento Django pode ser instalado por `pip`, através do comando `pip install Django`. A versão instalada é 3.2.5. O projeto foi preparado a partir do comando `django-admin startproject main`. Este comando cria o ficheiro `manage.py`, que se vai utilizar a partir deste momento para interagir com as operações Django. Nomeadamente, o comando `python manage.py runserver` inicia o servidor de desenvolvimento.

Para permitir a comunicação do servidor Django com a base de dados, foi instalada a biblioteca Python `psycopg2` a partir de `pip`.

4.7.2 Definições

O ficheiro `settings.py` associado à *framework* sofreu algumas alterações desde a sua génese.

Uma destas alterações inclui a substituição da base de dados por omissão SQLite pela PostgreSQL com PostGIS configurada na Secção 4.6 e os seus dados de autenticação.

Para além disso, foram adicionadas variáveis referentes ao correio eletrónico, de forma a enviar as notificações de eventos, e-mails de confirmação ou redefinir palavra-passe.

Foi definida uma diretoria para ficheiros estáticos como CSS e *favicon* dentro da pasta `main`. Dentro da mesma pasta foi definida a diretoria para os *templates* HTML.

4.7.3 Aplicações

No contexto Django, o termo projeto descreve uma aplicação *Web*. Na prática, descreve um pacote Python que fornece um conjunto distinto de

funcionalidades. A ideia que leva à criação de aplicações é a sua reutilização ao longo de vários projetos.

No contexto do projeto desenvolvido, foram separadas as funções ao longo de três aplicações personalizadas: `accounts`, `stations` e `events`. Cada uma destas aplicações trata do seu ramo específico de funções: `accounts` disponibiliza funcionalidades de registo e autenticação ao utilizador, `stations` trata das operações CRUD sobre estações e `events` das operações CRUD sobre eventos.

4.7.4 Modelo de dados

O modelo de dados proposto na Secção 3.4 foi implementado através de modelos Django.

Um modelo é a fonte única e definitiva de informação sobre os seus dados. Contém os campos e comportamentos essenciais dos dados que está a armazenar. Geralmente, cada modelo mapeia para uma única tabela de base de dados. Cada modelo é uma classe Python que deriva da classe do pacote `db` (base de dados) Django `Model`. Dentro dessa classe, cada atributo do modelo representa um campo de base de dados. Com tudo isto, a *framework* fornece uma API de acesso à base de dados gerada automaticamente.

O modelo de dados Django é sincronizado com a base de dados PostgreSQL através de migrações. Estas migrações são preparadas pelo comando `python manage.py makemigrations` e concretizadas pelo comando `python manage.py migrate`.

Note-se que para cada modelo que não inclua chave primária explícita é lhe atribuído um identificador numérico de formato `SERIAL` que se auto incrementa por cada novo objeto.

Na aplicação `accounts` está incluído um modelo implícito. O modelo `User`, disponibilizado em inteiro pelo Django, é suficiente para as necessidades do projeto. O único detalhe a ter em conta é que na interface *admin* disponibilizada não existe ativação de contas nem é obrigatório associar um endereço de correio eletrónico ao utilizador. Como tal, o foco da aplicação `accounts` será personalizar as páginas de *login* e registo nesse sentido.

Na aplicação `stations` são definidos dois modelos: `Station` e `Measure`.

O modelo `Station` foi dotado de seis atributos, os quais são descritos nos próximos parágrafos.

O seu "tópico" é representado por uma instância de `CharField` com oito caracteres de comprimento máximo. Este campo serve como chave primária e consequentemente identificador principal de cada estação. A vantagem do uso deste formato é disponibilizar ao utilizador uma identificação mais legível para reconhecimento das estações. Para além disso, o escalamento com o aumento do número de estações é mais suave dado que a combinação de letras é mais complexa que a combinação de números.

O formato designado para o tópico foi de três letras extraídas do início do nome completo da estação seguidos de um número (até 5 dígitos) que indica quantas estações utilizaram a mesma combinação antes da génese deste identificador. No caso extremo, existem $26^3 = 17576$ combinações só das primeiras três letras, e para cada uma destas 99999 números (o número 0 é excluído). Como tal, o total de combinações possíveis excede 1.750 mil milhões. Na prática, é normal que se chegue ao limite mais cedo, dado que a linguagem favorece algumas letras e sílabas sobre outras.

O "nome" da estação é um campo simples `CharField` com máximo de 100 caracteres. O campo `CharField` no Django corresponde ao formato `varchar` quando feita a migração para a base de dados PostgreSQL.

Para concretizar a conversão do nome em tópico foi necessário implementar uma função que limpasse a *string* e descobrisse o próximo número a incluir à frente das três letras. Para tal, foi criado um módulo Python personalizado `ids`. Este módulo está na sua íntegra representado no Código 4.10.

Código 4.10: Módulo responsável por gerar novos identificadores para as estações

```
1  import re
2  import unicodedata
3  from .models import Station
4
5  def strip_accents(text):
6      """
7      Strip accents from input String.
8
9      :param text: The input string.
10     :type text: String.
11
12     :returns: The processed String.
```

```
13      :rtype: String.
14      """
15      text = unicodedata.normalize('NFD', text)
16      text = text.encode('ascii', 'ignore')
17      text = text.decode("utf-8")
18      return str(text)
19
20  def text_to_id(text):
21      """
22      Convert input text to id.
23      """
24      text = strip_accents(text.lower())
25      text = re.sub('[ ]+', '', text)
26      text = re.sub('[^a-zA-Z-]', '', text)
27      return text
28
29  def name_to_topic(name, previous=None):
30      name = text_to_id(name)
31      prefix = name[:3].upper()
32      if prefix == previous[:3]:
33          return previous
34      topics =
        ↪  list(Station.objects.filter(topic__contains=prefix).values_list(
        ↪  'topic', flat=True))
35      if topics:
36          indexes = [ int(item.replace(prefix, "")) for item in topics ]
37          next_idx = sorted(indexes, reverse=True)[0] + 1
38      else:
39          next_idx = 1
40      return prefix + str(next_idx)
```

Os restantes campos de `Station` são a sua localização, instância da classe `PointField`, tomando partido das funcionalidades geográficas PostGIS, uma descrição opcional que é representada por um `TextField`, o dono da estação, que é uma chave estrangeira instância do modelo `User`, e uma fotografia opcional que é armazenada na diretoria `media` do projeto. Para o armazenamento desta fotografia foi ainda criada uma classe `OverwriteStorage` que deriva da classe Django `FileSystemStorage` com o objetivo de dar *override*

ao método `get_available_name` para que este remova uma imagem já existente com o mesmo nome ao substituir a imagem dessa estação por outra. Caso contrário, são guardadas todas as imagens carregadas e torna-se num ponto de vulnerabilidade do sistema.

Finalizando o modelo `Station`, este inclui também uma propriedade `get_absolute_url` que facilita a hiperligação para a página individual da estação em questão.

O modelo `Measure` é simples mas crucial. Os seus campos são `station` e `data`. O campo `station` é uma chave estrangeira que liga a medição a um objeto baseado no modelo `Station`, e o campo `data` é uma instância de `JSONField`. Para além disso, tem uma propriedade `timestamp` que potencia a leitura direta do valor incluído no JSON nos *templates*. Este modelo é responsável por guardar todas as medições vindas das estações recebidas pelos *broker* MQTT.

Na aplicação `events` consta apenas um modelo `Event` que armazena a condição e a sua ligação à estação e o seu criador. De acordo com a base de dados projetada na Secção 3.4 um evento seria disponibilizado ao longo de vários utilizadores. Durante a implementação optou-se por limitar os eventos aos seus criadores e às estações que os seus criadores gerem.

A condição de um evento é dividida por três partes como descrito na Secção 3.4. No caso do modelo os possíveis valores para os campos são representados por uma lista de tuplos `(id, nome)`, como é apresentado no Código 4.11:

Código 4.11: Listas de escolhas para os campos do modelo `Event`

```
SIGN_CHOICES = [
    ('lt', 'less than'),
    ('lte', 'less than or equal to'),
    ('eq', 'equal to'),
    ('gte', 'greater than or equal to'),
    ('gt', 'greater than'),
]

TYPE_CHOICES = [
    ("tmp", "Temperature"),
    ("hum", "Relative humidity"),
```

```
12      ("atm", "Atmospheric pressure"),
13      ("ws", "Wind speed"),
14      ("wd", "Wind direction"),
15      ("pre", "Precipitation"),
16  ]
```

Para além das associações e condição o modelo `Event` tem também dois campos `TimeField` que armazenam horas do dia. Um é a hora de começo e outro a hora de término. Isto potencia aos utilizadores escolherem uma intervalo do dia em específico para a condição se poder ativar.

Estes modelos foram registados na interface *admin* do Django de forma a que se evita ter de fazer variações das páginas para diferentes níveis de permissão.

4.7.5 Conceito de *templates*

No contexto Django, um *template* é um documento textual ou uma *string* Python assinalada com elementos da linguagem de *template* Django. Algumas estruturas são reconhecidas e interpretadas pelo motor. Destas, as principais são variáveis, etiquetas e filtros.

Um *template* é interpretado juntamente com um contexto. A renderização substitui as variáveis pelos seus respectivos valores, consultados a partir do contexto, e executa as etiquetas. Todo o resto é reproduzido como texto HTML.

Existe também a alternativa de optar por outra linguagem de *templates*, compatível com qualquer *framework* Python, de nome Jinja2. As diferenças não são significativas, e podem existir limitações quanto ao uso de etiquetas personalizadas, logo não justifica a troca.

A linguagem de *templates* Django inclui as etiquetas `extends` e `block` que são cruciais na redução da complexidade do conjunto de *templates* utilizados para apresentar o *website*.

Os templates `base`, `footer` e `header` foram criados de forma a funcionarem como base para todos os outros *templates*, limitando esses *templates* a incluir apenas o conteúdo específico à sua função. No *template* base foram incluídos os elementos *link* responsáveis por importar bibliotecas como

JQuery, o tipo de fonte a partir de Google Fonts e o *favicon* personalizado para o projeto.

Tanto para a aplicação `events` como para a aplicação `stations` foram criados *templates* `_list` e `_form` que apenas atualizam o título da página e o aspeto do conteúdo entre cabeçalho e rodapé.

No caso da página detalhada de estação foram implementadas etiquetas personalizadas, apresentadas no Código 4.12, para ler a partir do *template* a lista de medições.

Código 4.12: Etiquetas personalizadas para pesquisa em JSON

```
register = template.Library()

@register.filter(name='lookup')
def lookup(value, arg):
    if value:
        # JSONField value is stored in String form
        obj = json.loads(value)
        return obj.get(arg)
    # else return None

@register.filter(name="values")
def values(value, arg):
    if value: # queryset is not empty
        res = [json.loads(sta.data).get(arg) for sta in value]
        try:
            res = [float(r) for r in res] # convert to float if possible
        except:
            pass
        return res
```

4.7.6 Tailwind CSS

Tailwind CSS [25] é uma *framework* CSS focada na utilidade, repleta de classes como `flex`, `pt-4`, `text-center` e `rotate-90` que podem ser combinadas para construir qualquer design directamente no *markup* HTML.

Node.js é um *runtime* de JavaScript dedicado a construir aplicações *Web*

escaláveis. Este já estava instalado na máquina antes da realização do projeto desenvolvido. A versão instalada é a v15.3.0.

A ferramenta `npm` é um gestor de pacotes para JavaScript e o por omissão de Node.js. Em muitos sentidos é semelhante à ferramenta `pip` do Python.

Para começar a instalação foi criada uma pasta `jstoolchain` na raíz do projeto Django. De seguida, com a ferramenta `npm` do Node.js foram corridos os comandos `npm install tailwindcss postcss-cli autoprefixer` e `npx tailwind init`. Seguindo as instruções [26] foi obtida uma maneira de construir, ao correr o comando `npm run build`, um ficheiro CSS incluído na pasta de ficheiros estáticos Django de forma a facilitar a construção do design das páginas.

No ficheiro de configuração, o parâmetro `purge` ativo permite, sem compromissos do visual, que o tamanho em memória do ficheiro CSS passe de 3742 para 30 *kilobytes*. É importante servir o ficheiro de tamanho mínimo ao utilizador para que este não demore demasiado tempo a carregar a página e encha rapidamente o seu *cache*.

4.7.7 *Views*

O ficheiro `urls.py` detalha o mapeamento de endereços *Web* para o que o Django chama de *views*. No contexto Django, uma *view* é uma função ou classe encarregue de processar pedidos HTTP e retornar documentos como resposta.

O aspeto final da lista `urlpatterns` em `urls.py` encontra-se representado no Código 4.13:

Código 4.13: Lista de formatos de URL disponíveis no *website*

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('', s.home, name='home'), # all stations
    path('accounts/login/', a.UpdatedLoginView.as_view(), name='login'),
    path('accounts/logout/', a.logout, name='logout'),
    #path('accounts/password_reset/', a.password_reset,
    ↪ name='password-reset'),
    path('accounts/signup/', a.signup, name='signup'),
    path('accounts/activate/<uidb64>/<token>/', a.activate,
    ↪ name='activate'),
```

```
9     path('events/', e.event_list, name='event-list'),
10    path('events/add/', e.EventCreateView.as_view(), name='event-add'),
11    path('events/<int:pk>/update/', e.EventUpdateView.as_view(),
      ↪  name='event-update'),
12    path('events/<int:pk>/delete/', e.EventDeleteView.as_view(),
      ↪  name='event-delete'),
13    path('stations/', s.station_list, name='station-list'),
14    path('stations/search/', s.station_search, name='station-search'),
15    path('stations/add/', s.StationCreateView.as_view(),
      ↪  name='station-add'),
16    path('stations/<str:pk>/', s.StationDetailView.as_view(),
      ↪  name='station-detail'),
17    path('stations/<str:pk>/update/', s.StationUpdateView.as_view(),
      ↪  name='station-update'),
18    path('stations/<str:pk>/delete/', s.StationDeleteView.as_view(),
      ↪  name='station-delete'),
19    path('get_user_location', s.get_user_location,
      ↪  name='get_user_location'),
20 ]
21 urlpatterns += static(settings.MEDIA_URL,
   ↪  document_root=settings.MEDIA_ROOT)
22 urlpatterns += staticfiles_urlpatterns()
```

`a`, `s` e `e` são as aplicações `accounts`, `stations` e `events`, respetivamente.

Os formulários Django são classes que convertem os campos dos modelos para *widgets* que permitem ao utilizador inserir o valor respetivo ao tipo de dados no formulário. Por exemplo, o campo localização do modelo `Station` quando convertido em formulário resulta num mapa-mundo interativo onde o utilizador pode escolher um ponto geográfico. Todas as *widgets* tiveram de ser adaptadas através das classes internas `Meta` dos formulários de forma a cumprirem o design planeado e não ficarem com mau aspeto.

Para além disso em alguns dos casos foi necessário nos *templates* incluir elementos `script` que alteram as propriedades dos elementos do formulário.

Em `accounts`, as *views* incluem a função `signup`, que disponibiliza o formulário no caso do pedido ser `HTTP GET` e regista o novo utilizador caso esse pedido for `HTTP POST`, incluindo enviar um *e-mail* para esse utilizador para ativação da conta.

Para além desta, `accounts` responde aos pedidos dos links gerados para ativação da conta com a ativação da mesma caso sejam válidos, tem outra *view* que simplesmente dá *logout* ao utilizador, e finalmente um formulário de login que foi adaptado para incluir uma caixa de seleção "*Remember me*" que ao submeter o login altera o tempo de expiração da sessão. As *views* `signup` e `UpdatedLoginView` redirecionam o utilizador para a página inicial caso já esteja autenticado.

A aplicação `stations` inclui a *view* que mostra a página inicial com as 20 estações mais próximas do utilizador, ordenadas por distância. A distância é ainda anotada a cada linha resultante da *query* e convertida para quilómetros.

Para descobrir a localização estimada do utilizador é utilizada a biblioteca GeoIP2 [27]. Neste caso a biblioteca GeoIP2 corresponde o endereço IP do utilizador a coordenadas geográficas. A configuração da biblioteca requiriu o *download* das bases de dados `GeoLite-City` e `GeoLite-Country`.

Para obter o endereço IP do utilizador é analisado o cabeçalho do pedido HTTP. Primeiro, o campo `HTTP_X_FORWARDED_FOR`, que mesmo incluindo endereços *proxy* serão como itinerário até ao endereço real do utilizador. Em último recurso, também o campo `REMOTE_ADDR` contém um endereço, que neste caso arrisca ser um dos endereços *proxy*.

O mesmo módulo `views` tem uma função dedicada a ser chamada por pedido AJAX. Esta *view*, apresentada no Código 4.14, responde aos pedidos com um documento JSON com as coordenadas geográficas obtidas através da biblioteca GeoIP2. Esta função é utilizada no *template* do cabeçalho e consequentemente em todas as páginas do *website*.

Código 4.14: Funções responsáveis pela página inicial e localizar o utilizador

```
1  g = GeoIP2()
2
3  def __get_client_ip(request):
4      x_forwarded_for = request.META.get('HTTP_X_FORWARDED_FOR')
5      if x_forwarded_for:
6          ip = x_forwarded_for.split(',')[0]
7      else:
8          ip = request.META.get('REMOTE_ADDR')
9      return ip
10
```

```
11 def get_user_location(request):
12     try:
13         ip_address = __get_client_ip(request)
14         lat, lon = g.lat_lon(ip_address)
15         response = { 'lat': lat, 'lon': lon }
16         return JsonResponse(response)
17     except:
18         return JsonResponse({ 'lat': '??', 'lon': '??' })
19 
20 def home(request):
21     '''Closest 20 stations'''
22     try:
23         ip_address = __get_client_ip(request)
24         pnt = g.geos(ip_address)
25         station_list =
           ↪  Station.objects.annotate(distance=Distance('location',
           ↪  pnt)).order_by('distance')[:20]
26         for sta in station_list:
27             value = float(str(sta.distance).split(" ")[0])
28             sta.distance = D(m=value).km
29     except:
30         station_list = Station.objects.order_by("topic")[:20]
31 
32     context = {
33         'stations': station_list,
34     }
35     return render(request, 'stations/home.html', context)
```

Outra *view* do mesmo módulo utilizada em todas as páginas é a que processa a pesquisa por estações. Esta responde na sua maior parte como a página inicial exceto que o número de estações não é limitado por número mas pela correspondência com o termo pesquisado. Esse termo é procurado sem maiúsculas ou minúsculas nos valores de tópico, nome e dono de cada estação.

A última *view* em forma de função deste módulo é a de lista de estações, que utiliza um *template* diferente pois é dedicada a mostrar as estações pertencentes a um utilizador autenticado, com botões para criar e apagar

estações. Esta função tem um decorador `@login_required` de forma a enviar de volta para a página inicial quem não esteja autenticado ao aceder à página.

As operações criar, atualizar e apagar estação são concretizadas por *class-based views*. A particularidade do formulário de criação é que dá *override* à função `form_valid` de forma a recolher o utilizador do pedido para adicionar como dono da estação criada. Por outro lado, os formulários de atualização e remoção substitui a função `test_func` de forma a não permitir a entrada a utilizadores que não sejam donos da estação ou administradores.

Tanto o formulário de criação como o de atualização receberam alterações no *widget* de *upload* da fotografia de forma ao utilizador ver a imagem atual caso ela existir e também a que ele carregou antes de submeter. O código responsável por isto está apresentado no Código 4.15.

Código 4.15: Elemento *script* responsável pelas alterações aos campos do formulário da estação

```
function readUpload(input) {
        let url = window.URL.createObjectURL(input.files[0]);
        $('#current_photo').css('background-image', `url(${url})`);
}

$(document).ready( function() {
    // generated elements that aren't editable through the form class
    $("#id_location_div_map").addClass("w-full flex flex-col
    ↪ items-center");
    $("ol-viewport ol-touch").addClass("rounded-lg");
    $(".clear_features").addClass("mb-3 relative font-medium leading-6
    ↪ text-blue-300 transition duration-150 ease-out
    ↪ hover:text-white");
    $(".clear_features").attr("x-data", "{ hover: false }");
        $(".clear_features a").text("Clear selection");

        let a = $("#photo-clear_id").prev();
        let source = a.attr('href');

        a.html(`<div id="current_photo" class="rounded-lg w-2/3 flex
        ↪ bg-auto mx-auto" style="padding-top: 66.7%;
        ↪ background-image: url('${source}'); background-size: cover;
        ↪ background-position: center;">`);
```

```
18          a.parent().contents().filter(function() {
19                  return this.nodeType == 3
20          })[0].textContent = ""; // remove unnecessary text nodes
21
22          $("input[type=file]").attr("onchange", "readUpload(this)");
23  });
```

A *view* mais importante de todas é a `StationDetailView`, apresentada no Código 4.16. Esta classe está encarregue de dar o contexto para apresentar todos os detalhes sobre uma estação. As variáveis que esta *view* envia para o *template* incluem:

A diferença em horas para o fuso horário onde a estação está localizada. Esta informação é obtida através da biblioteca `pytzwhere` para a latitude e longitude guardadas na coluna "*location*". Para além desta informação envia também o número de milissegundos restantes até à meia-noite: este número serve para preparar um evento JavaScript que altera o dia apresentado à meia-noite no fuso horário da estação.

As coordenadas da estação são utilizadas também numa API grátis de "geocodificação inversa" para extrair do resultado a localidade e país onde a estação se localiza.

Finalmente, envia todos uma lista com o resultado do pedido à base de dados por todas as medidas pertencentes à estação em questão, ordenadas inversamente por marca temporal, para que a primeira seja a mais recente.

Código 4.16: Classe responsável por disponibilizar o contexto à página de detalhe da estação

```
1  class StationDetailView(DetailView):
2      model = Station
3      template_name = 'stations/report.html'
4
5      def get_context_data(self, **kwargs):
6          context = super().get_context_data(**kwargs)
7          lon, lat = self.get_object().location
8
9          # LOCAL DATE GET
```

```
10          tzw = tzwhere.tzwhere()
11          local_time = datetime.now(tz.gettz(tzw.tzNameAt(lat, lon)))
12          next_midnight = datetime.combine(local_time,
            ↪  datetime.max.time()) + timedelta.resolution
13          millis_until = (next_midnight.timestamp() -
            ↪  local_time.timestamp()) * 1000
14          context['tzOffset'] = local_time.utcoffset().seconds/3600
            ↪  #.timestamp() * 1000
15          context['untilMidnight'] = millis_until
16
17          # CITY/COUNTRY GET
18          url = 'https://api.bigdatacloud.net/data/reverse-geocode-client'
19          params = {'latitude': lat, 'longitude': lon, 'localityLanguage':
            ↪  "en"}
20          r = requests.get(url, params=params)
21          result = r.json()
22          context['city'] = result['locality']
23          context['country'] = result['countryCode']
24
25          measure_list = Measure.objects.filter(station=self.get_object())
26          measure_list = sorted(measure_list, key=lambda a: a.timestamp,
            ↪  reverse=True)
27          context['measures'] = measure_list
28
29          return context
```

A aplicação `events` tem *views* muito semelhantes às de *stations* adaptadas para eventos, sendo que a grande diferença é que cada evento não tem uma página dedicada.

Note-se que foi necessário adicionar um filtro ao *dropdown* para a estação do formulário de criação e remoção de eventos de forma a aparecerem apenas estações que pertençam ao utilizador.

4.7.8 *Widgets* Javascript

A página detalhada da estação inclui um mapa interativo centrado na localização da mesma com um *pin*. Para concretizar o mapa foi utilizada a biblioteca OpenLayers para Javascript. Optou-se pela mesma para man-

ter consistência entre esta e a disponibilizada pelo Django no formulário de criação e edição de estação. O código necessário para incluir o mapa na página inclui um elemento `div` com `id="map"` e `class="map"` e um objeto Javascript, apresentado no Código 4.17:

Código 4.17: *Script* responsável por renderizar o mapa na página de detalhe da estação

```
ol.proj.useGeographic();
{% autoescape off %}
let point = {{ station.location.geojson }};
{% endautoescape %}
const center = point.coordinates;
const feature = new ol.Feature({
    geometry: new ol.geom.Point(center),
    name: '{{ station.name }}'
});

var map = new ol.Map({
    controls: [],
    target: 'map',
    layers: [
    new ol.layer.Tile({
        source: new ol.source.OSM()
    }),
    new ol.layer.Vector({
        source: new ol.source.Vector ({
            features: [feature]
        }),
        style: new ol.style.Style ({
            image: new ol.style.Icon({
                src: '{% static 'icon/map_marker.png' %}',
                scale: 0.08,
                anchor: [0.5, 1.0],
                anchorXUnits: 'fraction',
                anchorYUnits: 'fraction',
                opacity: 0.85
            })
        })
    })
```

```
    ],
    view: new ol.View({
        //projection: 'EPSG:4326',
        center: center,
        zoom: 16
    })
});
```

Chart.js é uma biblioteca JavaScript *open-source* dedicada à visualização de dados em gráficos. Quando combinada com Moment.js, uma biblioteca JavaScript que facilita o processo de analisar, validar, manipular e exibir em formato data/hora, Chart.js permite a visualização dos dados num gráfico de eixo temporal.

As etiquetas personalizadas desenvolvidas em 4.7.5 foram utilizadas para preencher *datasets* como demonstrado no Código 4.18.

Código 4.18: *Script* responsável por renderizar o gráfico da temperatura na página de detalhe da estação

```
const canvas = document.getElementById('timeSeriesChart');
const ctx = canvas.getContext('2d');

{% autoescape off %}
let labels = {{ measures|values:"timestamp" }}.reverse();
let dates = labels.map(str => new Date(str));
{% endautoescape %}

const chart = new Chart(ctx, {
    type: 'line',
    data: {
    labels: dates,
    datasets: [{
        label: 'Temperature',
        data: {{ measures|values:"tmp" }}.reverse(),
        borderWidth: 1,
    }]
    },
    options: {
```

```
        responsive: true,
        maintainAspectRatio: false,
        scales: {
            xAxes: [
                {
                    display: true,
                    type: 'time',
                    time: {
                        parser: 'MM/DD/YYYY HH:mm',
                        tooltipFormat: 'll HH:mm',
                        unit: 'day',
                        unitStepSize: 1,
                        displayFormats: {
                            'day': 'D MMM'
                        }
                    }
                }
            ]
        }
    }
});
```

4.8 Subscritor MQTT

Para recolher os dados dos *broker* MQTT foi implementado um comando personalizado Django. Desta forma, a recolha de dados é independente do serviço das páginas *Web*. Para ativar a recolha de dados do *broker* corre-se numa nova linha de comandos o comando `python manage.py collectdata`.

Ao ser chamado o comando cria uma *thread* por cada *broker* disponível. Cada *thread* consiste num ciclo em que o cliente MQTT aguarda novas mensagens e processa-as com a função `on_message`, apresentada no Código 4.19.

Código 4.19: Função *callback* para receção de mensagens MQTT

```
def on_message(self, client, userdata, msg):
    pk = msg.topic.split("/")[-1]
```

```
3       measure = msg.payload.decode()
4       Measure.objects.register(station=Station.objects.get(topic=pk),
        ↪  data=measure)
5       self.stdout.write("New measure registered from " + pk)
6
7       obj = json.loads(measure)
8       for type in obj.keys():
9           # check type
10          event_list =
            ↪  Event.objects.filter(station=Station.objects.get(topic=pk),
            ↪  measure_type=type)
11
12          for ev in event_list:
13              # check condition
14              sign = ev.condition_sign
15              condition = False
16              value = float(obj[type])
17              if sign == 'lt':
18                  condition = value < ev.condition_value
19              elif sign == 'lte':
20                  condition = value <= ev.condition_value
21              elif sign == 'eq':
22                  condition = value == ev.condition_value
23              elif sign == 'gte':
24                  condition = value >= ev.condition_value
25              elif sign == 'gt':
26                  condition = value > ev.condition_value
27
28              # check time slot
29              if ev.time_start < datetime.now().time() < ev.time_end and
                ↪  condition:
30                  send_event_mail(ev)
31                  self.stdout.write("Notified " + ev.creator.username + "
                    ↪  of event")
```

A função apresentada no Código 4.19 tem também a responsabilidade de enviar *e-mails* de notificação ao utilizador caso em alguma medição se verifique que algum evento passe a todas as verificações. A função `register`

utilizada ao inserir novas medições evita que sejam inseridas medições duplicadas.

4.9 Previsão meteorológica

Para a escolha do modelo matemático a utilizar foram tidos em consideração os resultados dos artigos mencionados no capítulo 2.

Utilizando dados obtidos em formato CSV do servidor FTP de *National Centers for Environmental Information* dos Estados Unidos para Austin, Texas, apresentado na Figura 4.7, isolaram-se os dados da temperatura.

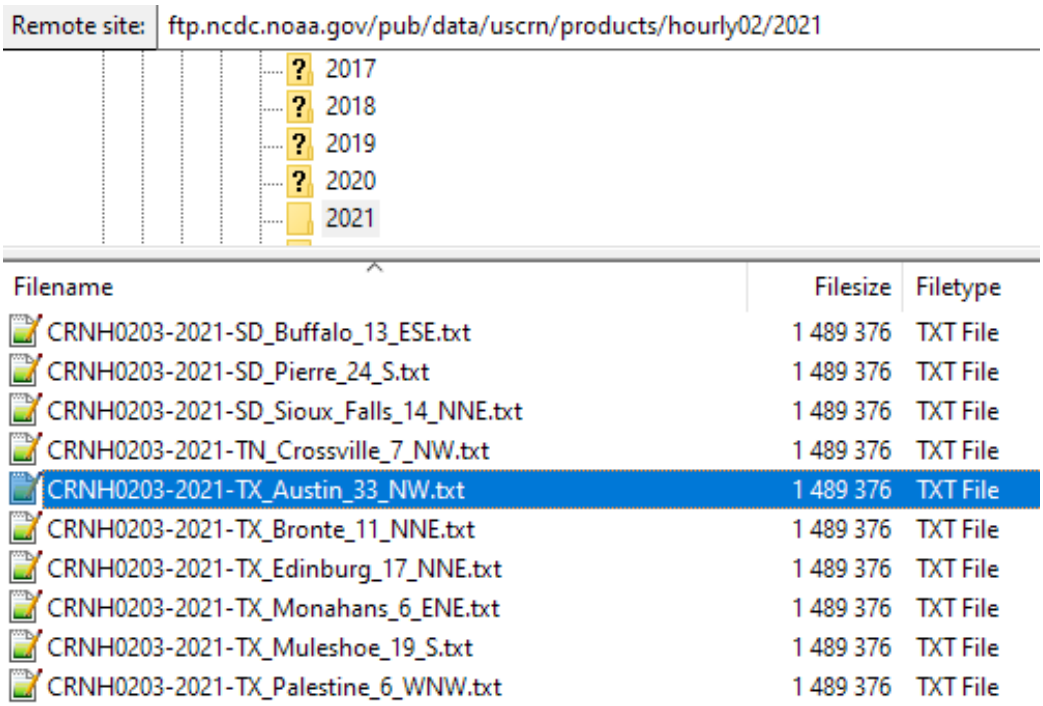

Figura 4.7: Aplicação FileZilla na diretoria de onde foram extraídos os dados

Selecionada uma semana, foram separados os primeiros seis dias para conjunto de treino e o último para conjunto de teste. Treinado o modelo ARIMA, foram comparados os valores reais e previstos da temperatura no dia 6 de abril. Os gráficos resultantes estão apresentados na Figura 4.8.

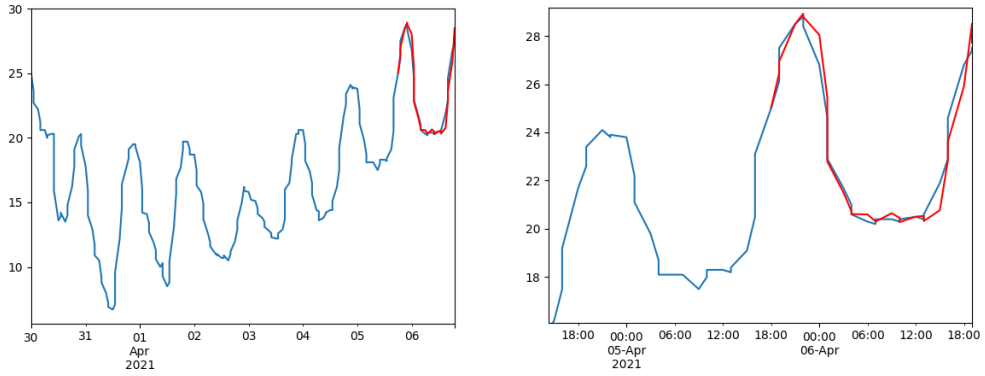

Figura 4.8: Resultados gráficos da previsão da semana de 30 de março a 6 de abril

Capítulo 5

Validação e Testes

Neste capítulo segue-se vagamente a ordem os diagramas de casos de utilização descritos na Secção 3.3 para explicar o resultado de cada função e ainda os detalhes adicionais que não constam do planeamento.

5.1 Estação

Para identificar as redes Wi-Fi disponíveis e os endereços I2C dos sensores instalados foram feitos testes na REPL do MicroPython através da ligação serial para a porta `COM3`, identificada pelo computador como a ligação série para o microcontrolador. Os resultados destes testes estão apresentados nas Figura 5.1.

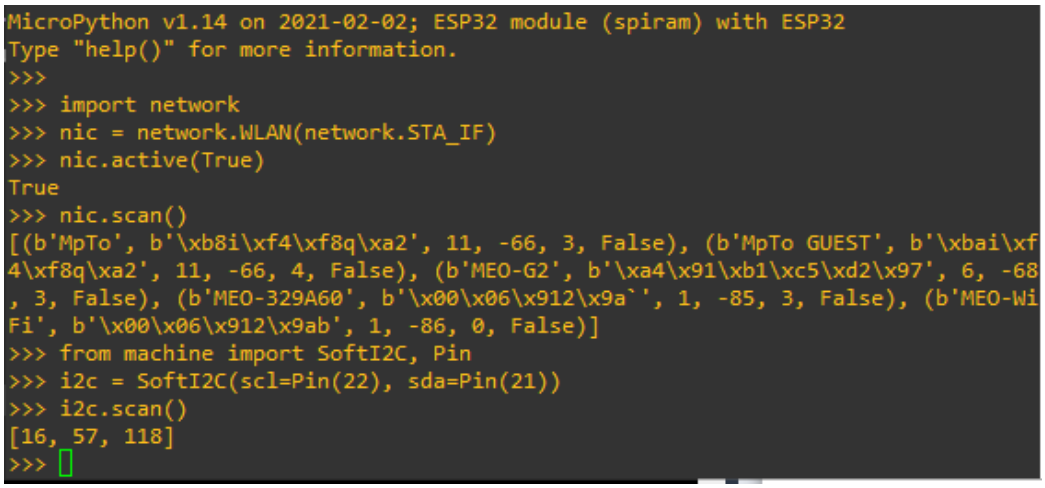

Figura 5.1: Resultados no REPL dos testes a I2C e Wi-Fi

Os testes ao código executado a partir do módulo `main` implicam a execução do mesmo através do comando Python `exec(open('main.py').read())`. Os resultados no REPL do mesmo são apresentados na Figura 5.2.

```
>>> exec(open('main.py').read())
Local time before synchronization: (2000, 1, 1, 0, 9, 17, 5, 1)
Local time after synchronization: (2021, 9, 12, 15, 6, 17, 6, 255)
Successfully connected to WLAN.
Connection to MQTT failed...
Min delay: 20.0 ms
Registered 0.0 rotations in 10 secs (0.0 rps).
Registered 0.0 rotations in 10 secs (0.0 rps).
{"tmp": "26.86", "hum": "56.81", "atm": "1006.56", "timestamp": "2021/09/12 15:0
6:37", "ws": "0.0"}
Registered 0.0 rotations in 10 secs (0.0 rps).
{"tmp": "27.01", "hum": "56.60", "atm": "1006.28", "timestamp": "2021/09/12 15:1
1:36", "ws": "0.0"}
Registered 23.75 rotations in 10 secs (2.375 rps).
{"tmp": "27.08", "hum": "56.56", "atm": "1006.19", "timestamp": "2021/09/12 15:1
6:35", "ws": "16.11637"}
```

Figura 5.2: Resultados no REPL da execução de `main` depois de 15 minutos

Na última medição da Figura 5.2 foi múltiplas vezes premido o botão de forma a confirmar o funcionamento do mecanismo do anemómetro. Alguns dos *prints* não parecem corretos, nomeadamente a linha "`Connection to MQTT failed...`" (deve-se ao código de retorno da conexão MQTT não estar implementado como consta na documentação) e a medição das rotações repetida. Estes erros não afetam o bom funcionamento da medição e envio para o *broker*, como será verificado na Secção 5.2.

5.2 MQTT

Os conteúdos do *broker* podem ser consultados através da aplicação MQTT Explorer [28]. A Figura 5.3 demonstra o tópico para onde foram enviadas as medições com uma mensagem retida. A interface identifica QoS como 0 mas foi testada a publicação e concluído que esse problema não advém da implementação MicroPython.

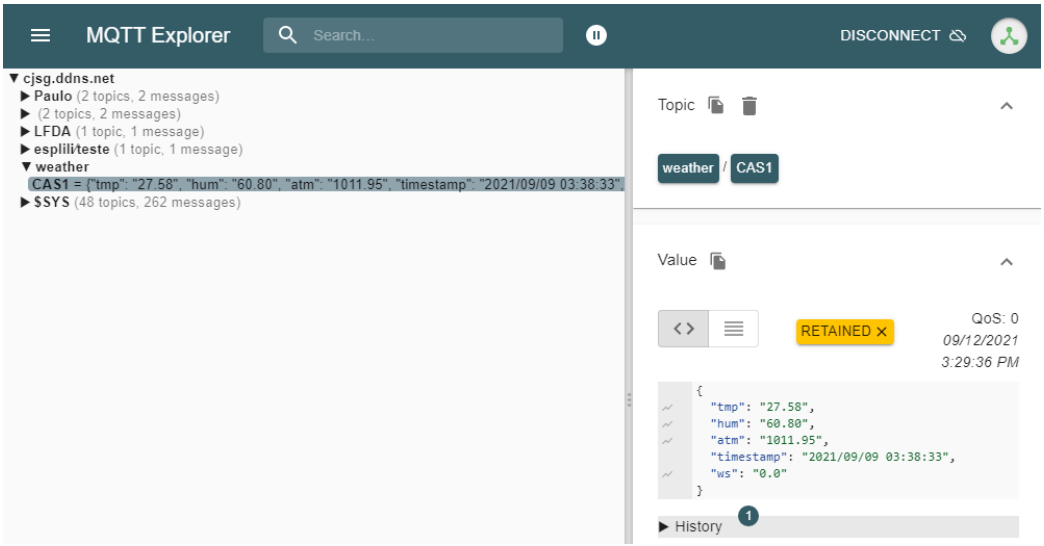

Figura 5.3: Janela de MQTT Explorer a mostrar os conteúdos do *broker*

5.3 *Website*

Ao *website* inteiro foi dado um visual *dark mode* com gradientes de verde a azul para ênfase.

5.3.1 Cabeçalho

O cabeçalho segue o planeado nos *mockups* da Secção 3.5. Os resultados com utilizador autenticado e não autenticado estão representados na Figura 5.4.

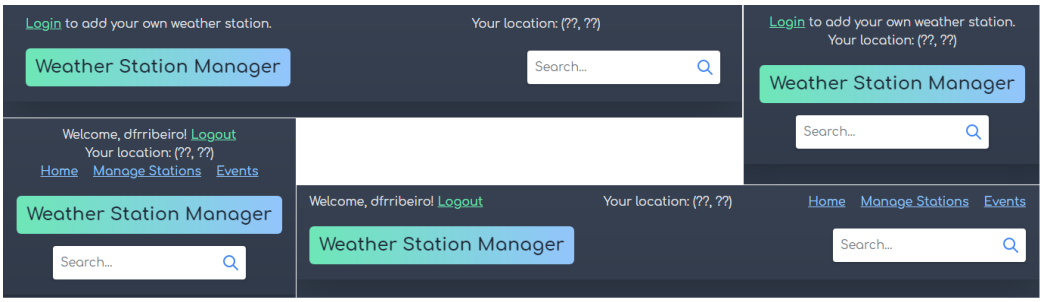

Figura 5.4: Montagem do cabeçalho da página nos formatos *Desktop* e móvel e com e sem sessão ativa

5.3.2 Rodapé

O rodapé da página foi simplificado em relação ao *mockup*. Como a Figura 5.5 apresenta, inclui hiperligações com ícones SVG que vão dar às páginas do FEIM 2021, ISEL, IPL, repositório GitHub do projeto desenvolvido e LinkedIn pessoal do autor.

Figura 5.5: Rodapé da página

Ao longo de todo o *website* as hiperligações azuis ou verdes têm uma animação de *fade* para branco ao passar o ponteiro do rato.

5.3.3 Página Inicial

A lista de estações apresentada pela página inicial tem as colunas tópico, nome, dono, latitude, longitude e distância ao utilizador. Ao reduzir a largura da página, de forma a manter uma disposição semelhante, são escondidas todas as colunas exceto tópico e distância. O seu aspeto completo encontra-se na Figura 5.15.

5.3.4 Páginas de autenticação

As páginas de autenticação apresentadas nas Figuras 5.6 e 5.7 não diferem muito do planeado nos *mockups*. O botão de visibilidade foi implementado manualmente a partir de JavaScript.

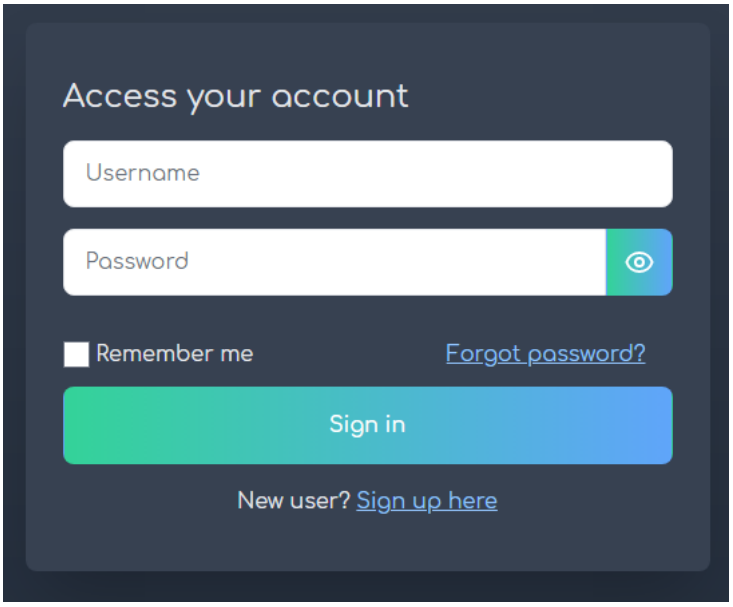

Figura 5.6: Conteúdo da página de *login*

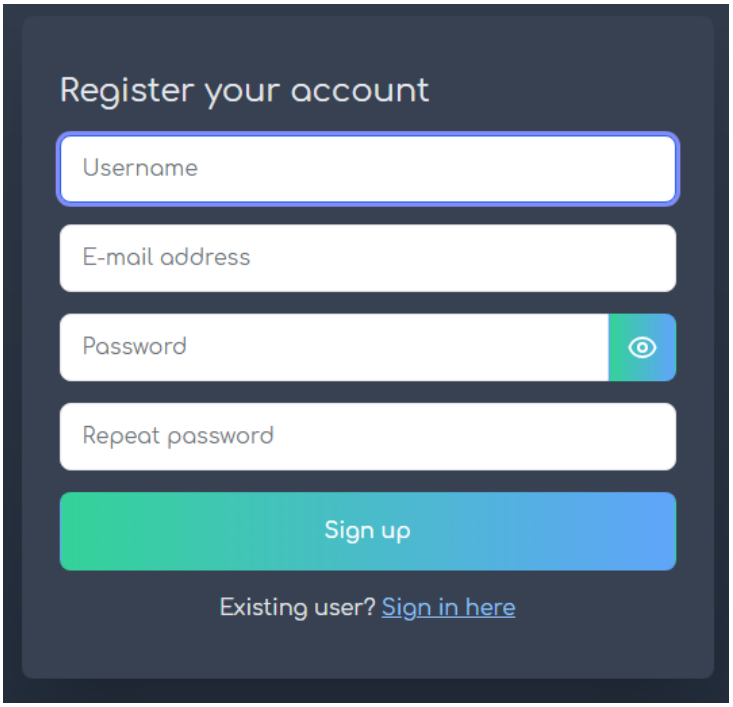

Figura 5.7: Conteúdo da página de registo

Depois do registo é recebido um *e-mail* para ativação da conta, como

demonstrado na Figura 5.8.

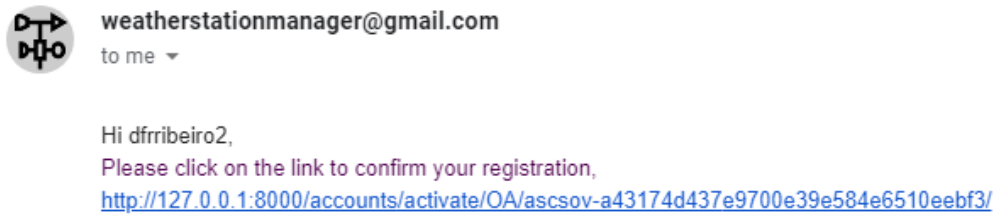

Figura 5.8: *E-mail* de ativação

Aquando a publicação, detalhada na Secção 5.4, a variável para o *host* vai derivar do pedido `HTTP POST` e como tal vai ser corrigido para um endereço IP público.

5.3.5 Lista de estações do utilizador

A Figura 5.9 demonstra a lista de estações do utilizador, que em relação à página inicial dispensa a coluna "dono" e acrescenta botões para criar e eliminar estações. Verifica-se que a página é de acesso proíbido para quem não esteja autenticado.

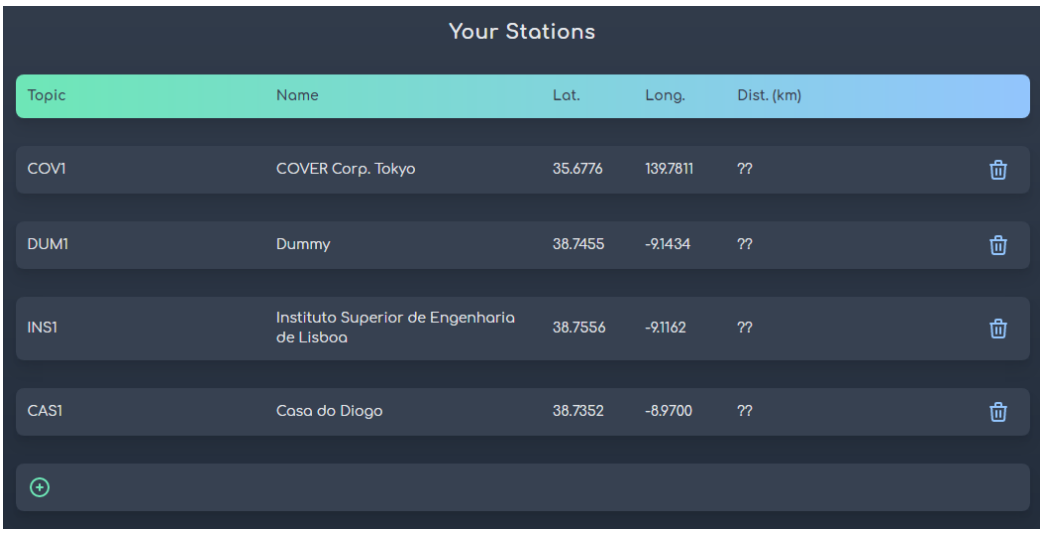

Figura 5.9: Aspeto da lista de estações do utilizador

5.3.6 Página de detalhe da estação

A página de detalhe da estação, demonstrado seu aspeto para o dono da estação em questão na Figura 5.10, apresenta um mapa, uma imagem representativa da estação sobreposta pela informação mais geral da mesma como dia, cidade e descrição, uma lista de informações sobre a última medição, uma linha de previsões não implementada, instruções para o envio de dados para o sistema e finalmente um gráfico da temperatura ao longo do tempo. Ao reduzir a largura da página os grupos horizontais de elementos passam a ser dispostos verticalmente.

O botão de edição da estação e instruções não aparecem caso o utilizador não esteja autenticado como dono da estação.

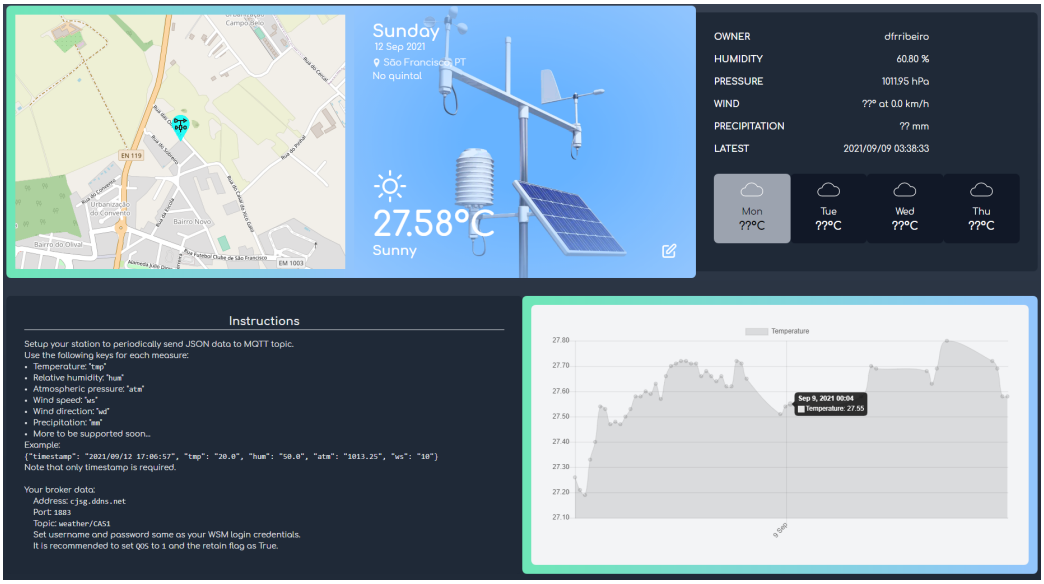

Figura 5.10: Aspeto da página de detalhe da estação

5.3.7 Lista de eventos

A lista de eventos é apresentada ao utilizador como demonstrado na Figura 5.11.

Figura 5.11: Aspeto da lista de eventos do utilizador

5.3.8 Formulários CRUD

Os formulários para remoção de um objeto seja ele estação ou evento aparecem como um botão simples de confirmação, como apresenta a Figura 5.12.

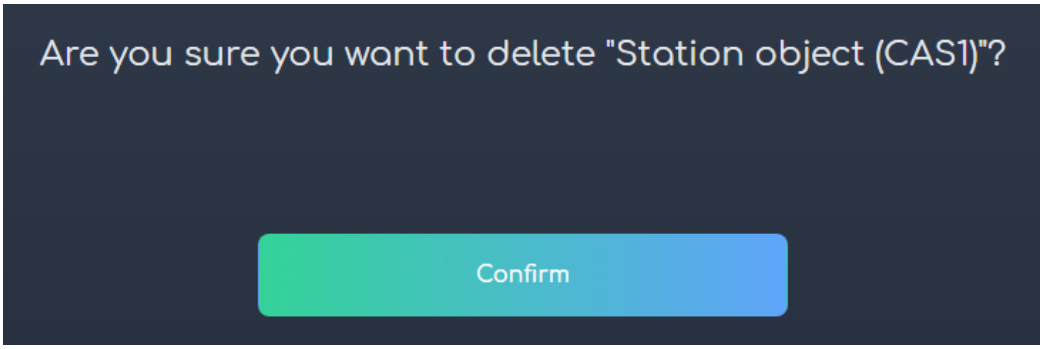

Figura 5.12: Confirmação de remoção de estação

Os formulários de criação e edição para ambos os objetos são efetivamente iguais, diferenciando-se apenas pela existência de dados pré-preenchidos. Para a estação têm o aspeto demonstrado na Figura 5.13 enquanto que para o evento está representado na Figura 5.14.

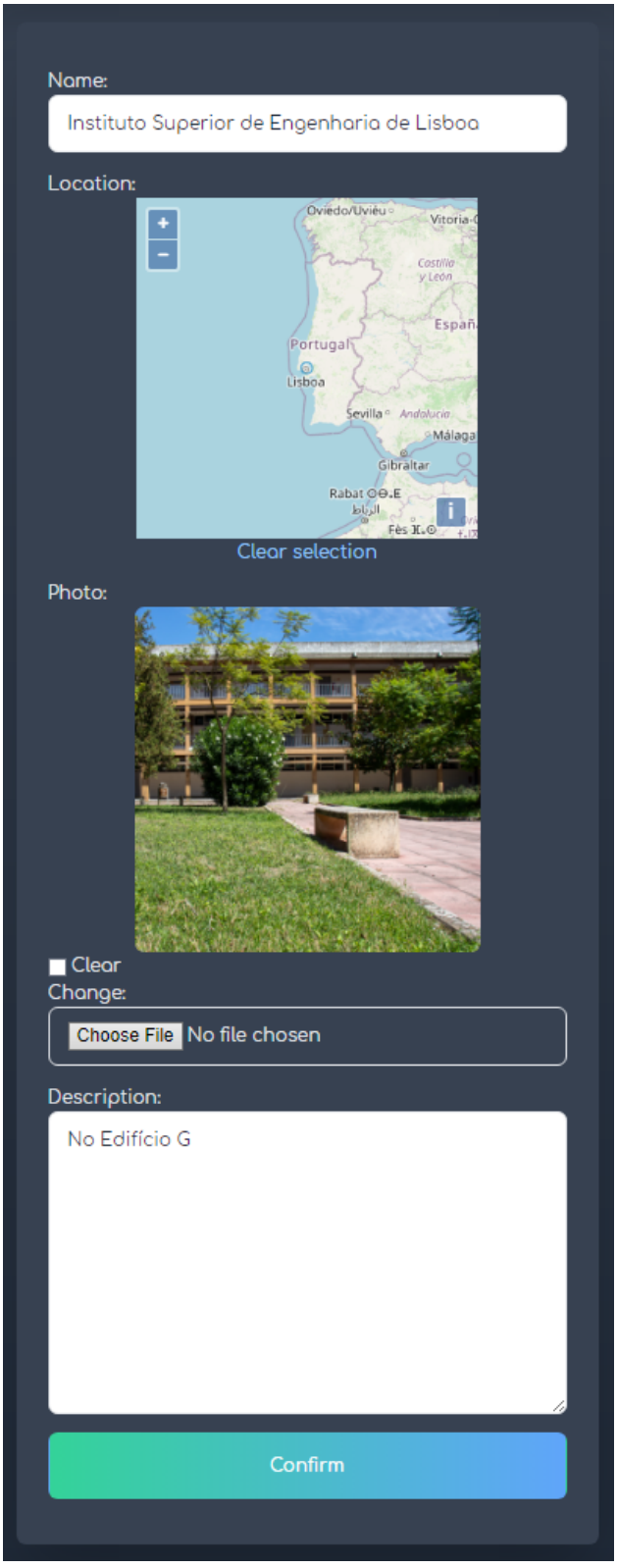

Figura 5.13: Formulário de edição de estação

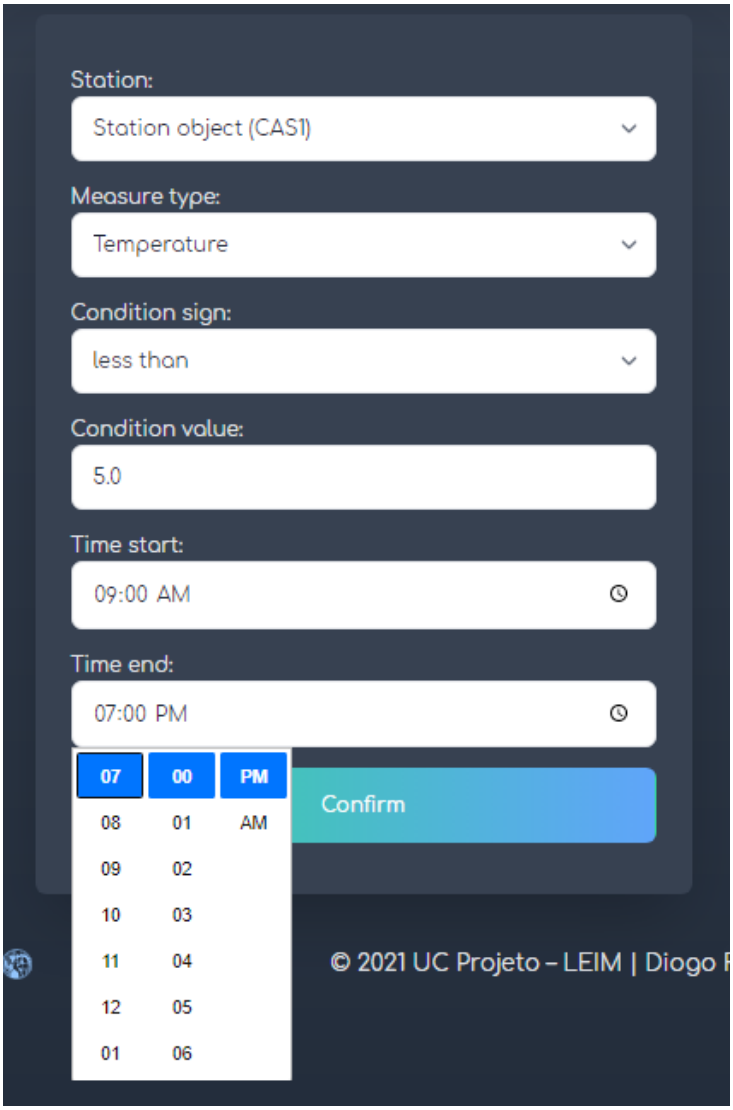

Figura 5.14: Página de edição de evento

5.4 Publicação do servidor

Apesar de ser com o servidor de desenvolvimento e não com o de produção como recomendado, foi arranjada uma maneira algo insegura para permitir acesso do exterior às páginas Web. Para tal, foi necessário adicionar uma regra na *firewall* Windows para aceitar ligações ao porto 8000 e no *router* foi adicionada uma regra NAT que mapeia o porto 8000 para o porto de saída 80.

Ligando o servidor no endereço 0.0.0.0:8000 com `ALLOWED_HOSTS=[*]` na configuração, a partir do momento em que a regra NAT fica ativa, o servidor é acessível a partir da Internet, algo que pode ser evidenciado desligando a Wi-Fi do telemóvel e acedendo ao site com dados móveis. O endereço do servidor é o IP público da rede em que se situa, que por acaso também suporta DNS dinâmico.

A publicação do servidor permite que seja verificado o funcionamento da geolocalização do utilizador e como tal preenchidas as distâncias nas listas de estações, sendo que se verifica uma precisão de um raio de cerca de 20 quilómetros.

O serviço Screenshot Machine [29] permite a captura da página inteira para diferentes formatos de ecrã. O resultado da captura da página inicial está apresentados na Figura 5.15.

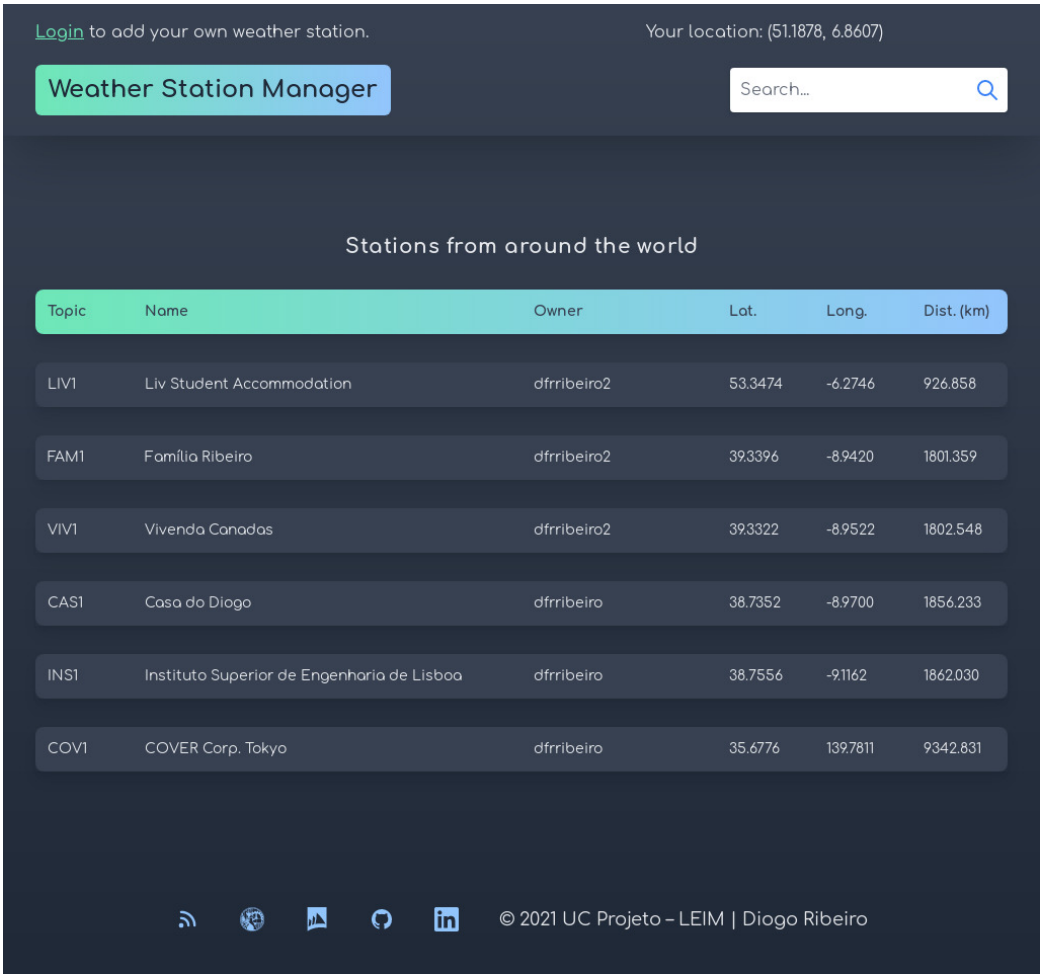

Figura 5.15: Aspeto da página inicial no formato *Desktop* capturado pelo serviço Screenshot Machine

Note-se que a localização no canto superior direito é associada ao utilizador que neste caso se situa perto de Düsseldorf na Alemanha.

Capítulo 6

Conclusões e Trabalho Futuro

O objetivo inicial do trabalho apresentado neste relatório consistia no desenvolvimento de um serviço de suporte a uma rede colaborativa de estações meteorológicas. Cada estação meteorológica é constituída por um microcontrolador, um módulo de comunicação e um conjunto de módulos de sensores que enviam os dados recolhidos para um servidor.

O grande volume do projeto na fase de planeamento evidencia uma desconexão entre a organização de tempo e os objetivos iniciais. No entanto, foi colocado um foco apertado em concretizar a ligação completa desde sensores a páginas *Web*. Nesse sentido, o sistema cumpriu maior parte das funções dos casos de utilização delineados na Secção 3.3.

Para além disso, o balanço é positivo para a flexibilidade do sistema, um dos requisitos não funcionais discutidos na Secção 3.1. O único requisito funcional não cumprido foram as previsões meteorológicas, que requereriam mais recursos incluindo tempo para integrar no projeto desenvolvido.

Quanto ao microcontrolador, para justificar a escolha do ESP32 em relação ao seu predecessor, ESP8266, foi tanto pela disponibilidade como para trabalho futuro, dado que não foram utilizados os pinos que o ESP32 tem a mais, nem as funcionalidades Bluetooth, Flash drive e SRAM. Tendo em conta os 24.5 *kibibytes* que os *scripts* de Python desenvolvidos ocupam, numa estação não muito mais complexa do que a desenvolvida poderia ser substituído, com folga, o ESP32 pelo seu predecessor ESP8266.

No futuro, poderá ser completada a integração do anemómetro e pluviómetro à estação prototipada, para além de melhorar a forma bloqueante como o anemómetro realiza as suas medições para utilizar o segundo *core* do proces-

sador do ESP32 através de *multithreading*.

No MQTT ficou em falta arranjar uma solução de autenticação no *broker* para a possível sabotagem entre utilizadores que utilizem tópicos de estações que não lhes pertencem.

Um plano futuro para os modelos `Station` e `Measure` seria acrescentar um campo para cada medição que correspondesse à chave JSON que a identifica. Desta forma, seria concebida ainda mais liberdade ao publicar mensagens a partir da estação. Ainda nesta vertente, poderia ser utilizada uma forma mais sofisticada de processamento da data ao ser recolhida pelo serviço meteorológico de forma a se interpretar vários formatos em vez forçar o formato `YYYY/MM/DD hh:mm:ss`.

Para além disso, poderiam ser incluídas na estrutura JSON as unidades de medida. A existência das mesmas permitiria, por exemplo, a medição da temperatura em graus Fahrenheit em vez de graus Celsius.

A consistência de unidades de medida nos dados permitiria então fazer os cálculos de variáveis derivadas como altitude, *heat index* ou *dew point*.

Quanto aos eventos ficou por concretizar uma lista que apresentasse um registo das suas ativações para consulta futura.

Todo o *website* é responsivo e como tal compatível com dispositivos móveis. Juntamente com o seu visual consistente inclui também algumas funcionalidades QOL que melhoram a ergonomia do sistema.

A obtenção da localização do utilizador pelo seu endereço IP poderia ser melhorado no caso de utilizadores móveis através do uso do sensor GPS do dispositivo móvel.

Na página de detalhe da estação poderia ter sido implementada uma forma de derivar o tipo de clima (sol, nublado, chuvoso, etc.) em vez do visual estático que este mantém atualmente. Para além disso seria ideal a página fazer periodicamente pedidos AJAX que atualizassem os dados em tempo real sem necessidade para atualizar a página.

Finalmente, ficou em falta o alojamento WSGI do *website* num servidor de produção como Apache.

Em conclusão, este projeto tem muito a melhorar, parte culpa da metodologia desatualizada e propensa a atrasos, concretizando no entanto uma base sólida para uma rede de colaboração meteorológica. Com este trabalho pretendeu-se acima de tudo mostrar que a gestão meteorológica não é exclu-

siva às organizações governamentais e que esta vai de mão-em-mão com o paradigma IoT.

Apêndice A

E-mails trocados com Stephan Siemen

Para o efeito de manter suficiente precisão científica nas previsões meteorológicas, e para resolver o problema de como fazer a previsão meteorológica em si, foram trocadas entre mim (o aluno) e um cientista do Centro Europeu para Previsões Meteorológicas de Médio Prazo (ECMWF) mensagens de correio eletrónico em inglês, que estão na sua totalidade dispostas nas páginas seguintes:

From: <46307@alunos.isel.ipl.pt>
To: <Stephan.Siemen@ecmwf.int>
Subject: My end of course project: Simple, short-range, local weather forecasting for a single weather station

Greetings.
I am an Informatics and Multimedia Engineering student at the Lisbon Superior Engineering Institute (ISEL).
I found you through this video [https://www.youtube.com/watch?v=3Brf6VM3TQA] where you did a presentation about Using Python in Weather Forecasting at the PyData London Meetup.

My End of Course Project is based around building a weather station.
Other than just collecting data from the sensors and displaying it, I figured it would be very interesting to use it for short-range (up to 48 hours) local (less than 2km range) weather forecasting. My findings in this field have ranged from trying to fit my data into an ARIMA time series model to searching about software like WXSIM and atmospheric models like WRF and CAM. One idea I've formed is that I'll need the biggest possible range of data (as to account for yearly variations and such), which I didn't quite account for given that my station is not yet live recording the data. This can prove to be a problem as I do not have infrastructure ready to support massive databases but it can also be worked around by using the excuse that time will pass and as such the station will get better forecasts over time.
I'm not so well versed in Meteorology, so I was looking for some premade model that I could just stuff my data into (from Python if possible). Furthermore, I'm questioning if satellite and radar data can be ignored or just mixed into my sensors' data from a public dataset.
I was trying to make this as simple as possible but also not straight up wrong, so I've clearly fallen into a rabbit hole. Do you mind recommending something for me to chase after, such as some documentation or an API I could use? Is it too ambitious for a two month project?

Thank you so much for your attention, and I'm sorry if I came barking up the wrong tree!

From: <Stephan.Siemen@ecmwf.int>
To: <46307@alunos.isel.ipl.pt>
Subject: Re: My end of course project: Simple, short-range, local weather forecasting for a single weather station

Dear Diogo,

Thanks for reaching out. To do any weather forecast from a single weather stations would be quite a challenge. Most weather forecast models will require data from a wider area to initialise the starting weather conditions. The complexity of setting up a full forecast model might exceed your time limit.

To do a very short weather forecast with a single weather stations you could train a regression model. This model could be used to make short time forecast. To train any model like that would require very long time series of data. There are various sources for long time series of weather records. The easiest would be a local record from a nearby weather station or university. Otherwise the ERA5 data set (https://www.ecmwf.int/en/forecasts/datasets/reanalysis-datasets/era5) is another option. You find examples of usage in the user documentation. The Copernicus user support will be able to help you if you have trouble using the data set.

Good luck with your project!
Best regards
Stephan

Dr Stephan Siemen

Head of Development Section
Forecast Department
European Centre for Medium-Range Weather Forecasts
Reading, UK | Bologna, Italy

Bibliografia

[1] D. Ribeiro. `weather-server` repository. Código-fonte do sistema desenvolvido. [Online]. Disponível em: https://github.com/dfrribeiro/weather-server

[2] smartbedded GmbH. Meteobridge. Visitado a 2021-06-01. [Online]. Disponível em: https://www.meteobridge.com/wiki/index.php/Home

[3] S. Loft. Cumulus. Visitado a 2021-06-01. [Online]. Disponível em: https://cumuluswiki.org/a/Main_Page

[4] Ministério do Mar. IPMA - Tempo - Previsão Localidade Horária. Visitado a 2021-06-01. [Online]. Disponível em: https://www.ipma.pt/pt/otempo/prev.localidade.hora/

[5] E. Casimiro. MeteoMoita. Visitado a 2021-06-01. [Online]. Disponível em: http://www.meteomoita.com

[6] Ministério do Mar. IPMA - Enciclopédia - Tempo - AROME. Visitado a 2021-06-01. [Online]. Disponível em: https://www.ipma.pt/pt/enciclopedia/otempo/previsao.numerica/index.html?page=arome.xml

[7] T. J. Ehrensperger. VXSIM Weather Simulator - Interactive Local Atmospheric Modeling System. Visitado a 2021-07-01. [Online]. Disponível em: https://www.wxsim.com/

[8] A. M. Gonçalves, C. Costa, M. Costa, S. O. Lopes, and R. Pereira, "Temperature Time Series Forecasting in the Optimal Challenges in Irrigation (TO CHAIR)," in *Computational Methods in Applied Sciences*, Department of Mathematics and Centre of Mathematics, Ed. Springer International Publishing, Novembro 2020, pp. 423–435.

[9] E. Abrahamsen, O. M. Brastein, and B. Lie, "Machine Learning in Python for Weather Forecast based on Freely Available Weather Data," in *Proceedings of The 59th Conference on imulation and Modelling (SIMS 59), 26-28 September 2018, Oslo Metropolitan University, Norway.* Linköping University Electronic Press, Novembro 2018.

[10] C. C. Raible, G. Bischof, K. Fraedrich, and E. Kirk, "Statistical Single-Station Short-Term Forecasting of Temperature and Probability of Precipitation: Area Interpolation and NWP Combination," vol. 14, no. 2, pp. 203–214, Abril 1999.

[11] W. W. Royce, "Managing the development of large software systems: concepts and techniques," in *Proceedings of the 9th international conference on Software Engineering*, ser. ICSE '87. Washington, DC, USA: IEEE Computer Society Press, 1987, p. 328–338.

[12] P. P.-S. Chen, "The entity-relationship model—toward a unified view of data," vol. 1, no. 1, pp. 9–36, Março 1976.

[13] Adafruit. ADAFRUIT BME280 I2C OR SPI TEMPERATURE HUMIDITY PRESSURE SENSOR. Visitado a 2021-09-01. [Online]. Disponível em: https://www.adafruit.com/product/2652

[14] Espressif Systems. ESP32 Series Datasheet. Visitado a 2021-07-01. [Online]. Disponível em: https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf

[15] D. George. MicroPython documentation. Visitado a 2021-07-01. [Online]. Disponível em: https://docs.micropython.org/en/v1.14/library/index.html

[16] S. Tatham. PuTTY - a free SSH and telnet client for Windows. Visitado a 2021-07-01. [Online]. Disponível em: https://www.putty.org/

[17] E. Gamma and Microsoft. Visual Studio Code - Code Editing. Redefined. Visitado a 2021-07-01. [Online]. Disponível em: https://code.visualstudio.com/

[18] R. Santos and S. Santos. MicroPython: BME280 with ESP32 and ESP8266 (Pressure, Temperature, Humidity). Visitado a 2021-09-01. [Online]. Disponível em: https://randomnerdtutorials.com/micropython-bme280-esp32-esp8266/

[19] R. Santos and S. Santos. MicroPython - Getting Started with MQTT on ESP32/ESP8266. Visitado a 2021-09-01. [Online]. Disponível em: https://randomnerdtutorials.com/micropython-mqtt-esp32-esp8266/

[20] M. Stonebraker. PostgreSQL: The World's Most Advanced Open Source Relational Database. Visitado a 2021-09-01. [Online]. Disponível em: https://www.postgresql.org/

[21] Refractions Research *et al.* PostGIS — Spatial and Geographic Objects for PostgreSQL. Visitado a 2021-09-01. [Online]. Disponível em: https://postgis.net/

[22] G. Formaro, P. Rottenberg, J. Hunter, and Django Software Foundation. GeoDjango Installation — Django Documentation. Visitado a 2021-09-01. [Online]. Disponível em: https://docs.djangoproject.com/en/3.2/ref/contrib/gis/install/

[23] G. Formaro, P. Rottenberg, J. Hunter, and Django Software Foundation. Django - the web framework for perfectionists with deadlines. Visitado a 2021-09-01. [Online]. Disponível em: https://www.djangoproject.com/

[24] A. Ronacher. Welcome to Flask - Web development, one drop at a time. Visitado a 2021-09-01. [Online]. Disponível em: https://flask.palletsprojects.com/en/2.0.x/

[25] A. Wathan. Tailwind CSS - Rapidly build modern websites without ever leaving your HTML. Visitado a 2021-09-01. [Online]. Disponível em: https://tailwindcss.com/

[26] "David D.". Python - How to use TailwindCSS with Django? Visitado a 2021-09-01. [Online]. Disponível em: https://stackoverflow.com/questions/63392426/how-to-use-tailwindcss-with-django

[27] G. Formaro, P. Rottenberg, J. Hunter, and Django Software Foundation. Geolocation with GeoIP2 - Django Documentation. Visitado a 2021-09-01. [Online]. Disponível em: https://docs.djangoproject.com/en/3.2/ref/contrib/gis/geoip2/

[28] T. Nordquist. MQTT Explorer - an all-round MQTT client that provides a structured topic overview. Visitado a 2021-09-01. [Online]. Disponível em: http://mqtt-explorer.com/

[29] Devtica s.r.o. Screenshot Machine - Take website screenshot with screenshot API. Visitado a 2021-09-01. [Online]. Disponível em: http://screenshotmachine.com/