RELATÓRIO DO PROJETO SOCIAL FIREFIGHTING

Universidade Autónoma de Lisboa

Orientador:

Professor Nuno Brás.

David Caçador nº20150735 Edgar Marchão nº20151092

Índice

Informações legais e preâmbulo

• Informações legais

Este projeto encontra-se sobre a licença GPL (*Gnu General Public License*) uma licença de desenvolvimento de software livres.

• Preâmbulo

Este projeto visa o desenvolvimento de uma aplicação móvel com o objetivo de permitir o envio de alarmes por cidadãos que a utilizem.

Esta aplicação móvel procura possuir as capacidades para o envio de alarmes (relatórios dos mesmos), por parte de cidadãos, de eventos reais para ajuda no combate aos incêndios, com ou sem imagem.

Para tal, necessita permitir que seja possível fazer o registo e o *login*, para identificação de qual o utilizador que está a enviar o alarme, principalmente devido á necessidade de identificar qual o nível de confiança no cidadão que enviou o alarme.

Finalmente, a aplicação móvel, procura exibir aos cidadãos todos os eventos, com os respetivos alarmes enviados para tal evento selecionado, aos cidadãos.

Visa também a classificação desses eventos como sendo reais ou não de modo a não fazer os bombeiros perderem tempo com alarmes falsos.

Visa ainda demonstrar aos bombeiros presentes no centro de comando cada alarme recebido por evento e a respetiva classificação do evento, possibilitando que estes decidam se intervêm em certo evento.

Procura finalmente que o método de classificação diminua os erros gerados por classificações erradas ao longo do tempo, através da classificação, por parte dos bombeiros, dos eventos.

Introdução

Objetivos

O projeto desenvolvido tinha como objetivos principais a construção de uma aplicação para dispositivo móvel (*front-end*), compatível quer em Android quer em IOS, ou seja, uma aplicação *cross-platform*.

Esta aplicação deveria ser capaz de enviar alarmes, reais ou não, sobre a existência de um incêndio na sua localização, por parte dos seus utilizadores. Este alarme deveria, também, possibilitar o envio de fotografias tiradas pelos cidadãos (os utilizadores da aplicação móvel) sobre o incidente.

Para tal acontecer é obrigatório efetuar as operações comuns a todas as aplicações: o registo do utilizador/cidadão na aplicação e criação da respetiva conta e a operação de *Login* para dar entrada como utilizador autenticado pelo sistema.

A aplicação tem também de ser capaz de mostrar os eventos/incêndios correntes aos utilizadores e permitir a sua seleção, que vai elaborar sobre esse mesmo evento, através da demonstração dos valores enviados, por cada alarme, pertencente ao evento selecionado.

Outro objetivo principal a atingir é o desenvolvimento de um servidor com eventos e alarmes associados através de uma base de dados (*back-end*), responsável pelo armazenamento dos dados transmitidos e pelas respostas a perguntas efetuadas pelo *front-end*.

Para a comunicação entre o *front-end* e o *back-end* poder acontecer em qualquer um dos sentidos é necessário a implementação de uma API, devido a estes serem programados em línguas de programação distintas e sendo uma prática geral da indústria falar através de APIs entre aplicações cliente/servidor.

Necessário a implementação de um modelo básico com o intuito de determinar a probabilidade de certo evento ser ou não um incêndio real. Para tal é necessário armazenar na base de dados os valores enviados pelos cidadãos quando estes enviam alarmes e, a partir destas variáveis executar uma previsão de probabilidade deste evento.

Estas variáveis são o nível de certeza do cidadão em o evento de que está a informar é mesmo incêndio, o ranking de confiança do próprio cidadão, consoante o número de alarmes que enviou e que foram classificados pelos bombeiros como terem sido efetivamente incêndios, o número de alarmes recebidos por cada evento, o nível de perigo á vida e aos edifícios que o circundam e o tipo de foto que o cidadão enviou com o alarme (se esta é uma fotografia de um incêndio ou não).

A partir destas operações implementadas no *back-end*, deve ser possível na aplicação apresentar os últimos eventos criados e os respetivos alarmes que os constituem ao utilizador da aplicação. Por fim, a aplicação tem de permitir fazer um *broadcast* para os cidadãos a informar sobre a existência de um incêndio na sua zona.

No projeto existe ainda dois objetivos extras a implementar.

Um desses objetivos é a determinação automática das fotografias recebidas através dos alarmes como apresentando um fogo ou como apresentando fumo ou não apresentar nenhum destes dois indícios.

O outro objetivo a implementar é um *front-end web* para fazer a apresentação dos eventos criados ao longo do decorrer da aplicação aos bombeiros e os respetivos alarmes enviados pelos

cidadãos. Permitindo que o bombeiro no comando possa analisar os dados recebidos e classificar o evento como sendo ou não, um evento real.

História

O nosso projeto vem tentar ajudar a combater os incêndios. Um Incêndio é uma ocorrência de fogo não controlado, que pode ser extremamente perigosa para os seres vivos e para as estruturas onde as pessoas habitam. A exposição a um incêndio pode levar à morte, geralmente pela inalação dos gases, ou pelo desmaio causado por eles, ou posteriormente pelas queimaduras graves.

Nem todos os fogos podem ser considerados incêndios, este é, no entanto, um tema que o senso-comum tem ao longo dos séculos banalizado de forma a que praticamente qualquer foco de fogo tem sido visto como "incêndio".

O Incêndio para ser caracterizado como tal tem que possuir certos fatores inerentes ao mesmo para ser considerado como tal. **[1]**

Alguns desses fatores são:

- A área ardida;
- As dimensões do rasto de destruição causado;
- A localização do mesmo.

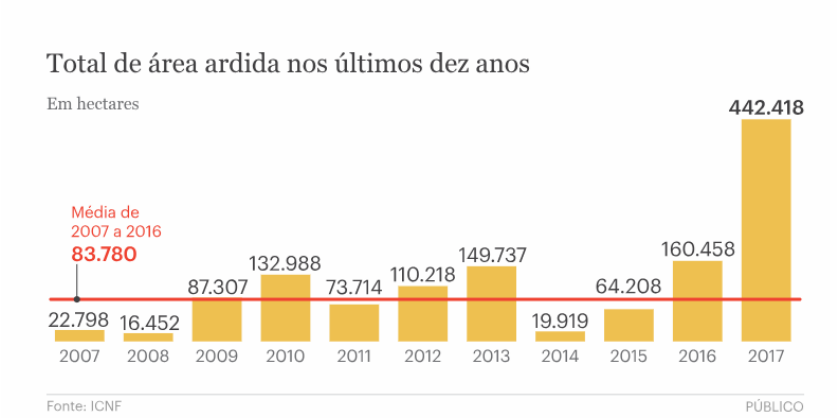
As normas sobre proteção de Incêndios classificam o risco que se apresenta em cada tipo de edifício segundo as suas características, para adequar os meios de prevenção.

O Risco atende a três fatores:

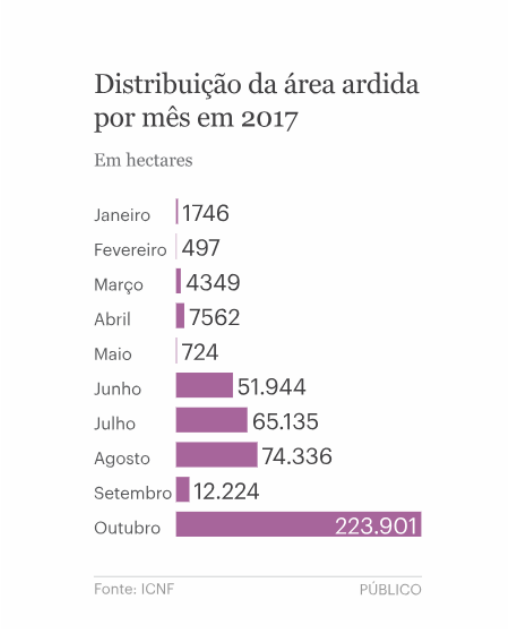
- Ocupação: maior ou menor quantidade de pessoas e o conhecimento que possuem os ocupantes do edifício;
- Composição: A construção do edifício em si, de que materiais é construído, qual é sua altura, etc;
- Conteúdo: Materiais mais ou menos inflamáveis, dentro do edifício, podem determinar o fator de risco de um incêndio.

Num ano marcado por mais de uma centena de mortes resultantes dos incêndios, foi divulgado que, entre o início do ano e o final do mês passado, arderam em Portugal mais de 440 mil hectares de floresta e povoamentos, o que corresponde a quatro vezes mais do que a média registada nos dez anos anteriores.

Isto significa que ardeu mais 428% do que arde, por norma, em anos anteriores (mais 358 mil hectares, totalizando um número superior à área total do distrito de Coimbra), o maior número registado desde 2006. Segundo cálculos feitos pelo PÚBLICO com base nos dados do sistema português, mais de 34% da área destruída nos últimos dez anos em Portugal devido a incêndios florestais ardeu durante este ano.



Em média, nos anos anteriores, a área ardida em mato era superior à dos povoamentos (62% de mato ardido e 38% de povoamentos) – mas este ano o número de área ardida em povoamentos foi maior. Arderam mais de 260 mil hectares em povoamento (59,9% do total ardido) e cerca de 177 mil hectares arderam em matos (aproximadamente 40% do total).



2017 foi precisamente marcado pela perda de vidas humanas resultantes de incêndios, um número sem precedentes em anos anteriores: no total, morreram mais de 100 pessoas. Os incêndios de Pedrógão Grande, em junho, fizeram 64 mortos (e ainda a morte de uma mulher que fugia do fogo), registando-se também mais de duas centenas de feridos; já em outubro, morreram 45 pessoas, contabilizando-se cerca de 70 feridos. **[2]**

Revisão Teórica

1. Back-end vs Front-end

Em qualquer aplicação que se pretenda desenvolver, existem duas grandes áreas de desenvolvimento ao qual ela se subdivide: ***front-end* e *back-end*.**

Se uma linguagem for classificada como *front-end*, significa que ela lida com a **interface de um utilizador**. Em outras palavras, quando um utilizador acede a um site ou a uma aplicação mobile, o é visível foi feito com a utilização de tecnologias *front-end*. Tento isto em conta o *front-end* também conhecido como *client-side*.

Para fornecer um exemplo, utilizaremos uma página feita em HTML. Uma página de HTML com títulos, imagens e vídeos por exemplo foi tudo programado em HTML, ou seja, todo o conteúdo presente na página é inserido com a utilização de códigos HTML, em forma de *tags*.

Apenas o HTML não é suficiente, é preciso fornecer um estilo à página. Quem concede essa identidade para uma página é o **CSS**. De volta ao exemplo da página: A cor de fundo da página, a formatação do texto, do cabeçalho, do rodapé e dos títulos, existe graças ao CSS.

Por último, temos o **JavaScript.** O responsável por trazer "vida à página". É através do JavaScript que a página ganha um comportamento: *sliders*, animações, efeitos de *scroll*, ações no site baseadas em eventos, entre outros. Agora sim, com a adição do JavaScript, a este conjunto podemos dizer que estamos perante uma programação *front-end*!

Se o *front-end* trata das interfaces para o utilizador, o *back-end* é aquele que fornece as informações para o *front-end*, ou seja, quando o utilizador faz login numa aplicação por exemplo, é o *back-end* que processa o seu endereço e a sua password e envia os dados para o *front-end* poder apresentá-los na página. O *back-end* é também conhecido por *server-side*.

Por ser uma área muito abrangente, existem diversas linguagens, frameworks e tecnologias para estudar. Cada um deles é focado em diferentes aspetos e soluções.

Existem tecnologias de *back-end* voltadas para web, como por exemplo o PHP, o Ruby on Rails o, Node.js ou o Python e existem também as linguagens focadas em soluções desktop, como o C, o C#, o C++ e o Java. **[3]**

Tal como a integração e utilização de ferramentas MySQL, Oracle e bases dados SQL são necessárias para que os processos das aplicações web sejam encontradas, gravadas e enviadas de volta para os utilizadores. **[4]**

2. O que é uma API

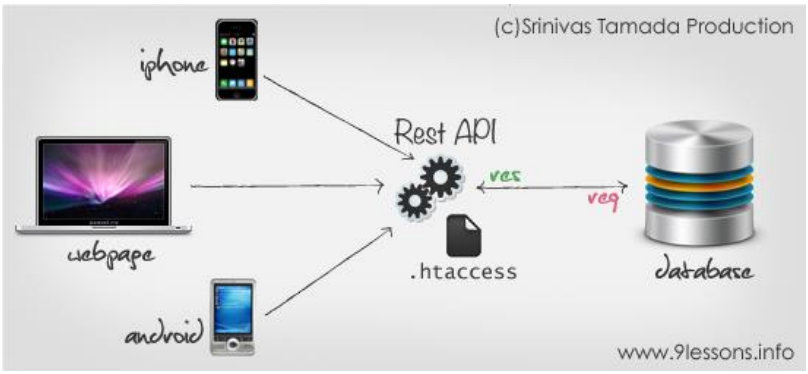
API é um conjunto de rotinas e padrões de programação para acesso a uma aplicação de *software* ou a um *Web application*. A sigla API refere-se ao termo em inglês "*Application Programming Interface*" que traduzindo à letra significa "Interface de Programação de Aplicações".

Uma API é criada quando uma empresa de software tem a intenção de que outros criadores de software desenvolvam produtos associados ao seu serviço. Existem vários deles que disponibilizam os seus códigos e instruções para serem usados em outros sites da maneira mais conveniente para os seus utilizadores.

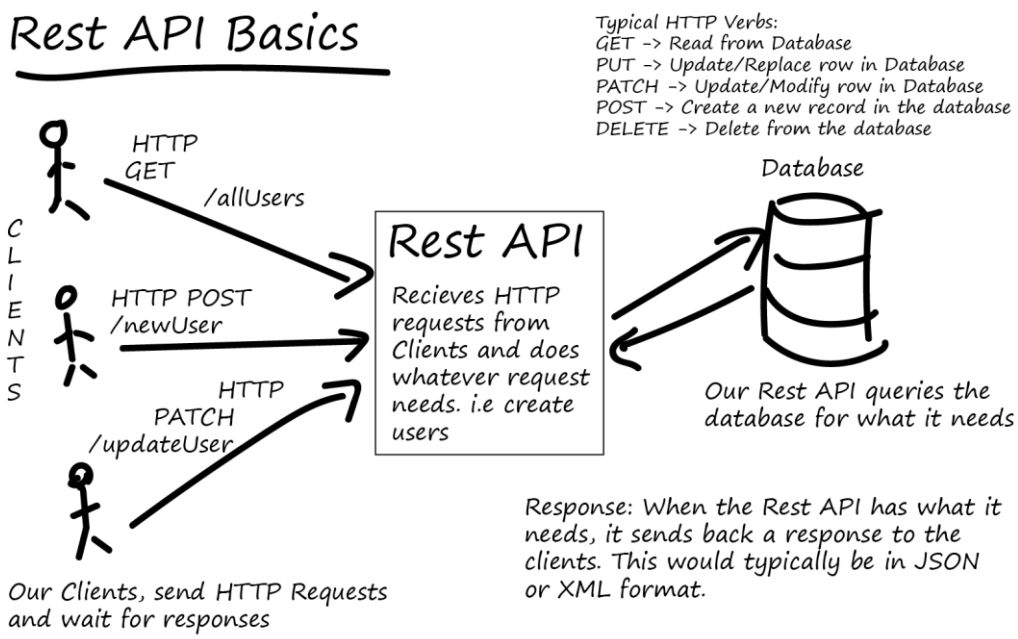
Através das APIs, as aplicações podem comunicar umas com as outras sem conhecimento ou intervenção dos utilizadores.

Elas funcionam através da comunicação de diversos códigos, definindo comportamentos específicos de determinado objeto numa interface. A API liga as diversas funções em um site de maneira que possam ser utilizadas em outras aplicações. **[5]**

API, Base de Dados e React Native



[6]



[7]

A REST API serve para receber HTTP *requests* dos clientes. Esses *requests* no caso da nossa aplicação podem ir desde criar uma conta para um utilizador como enviar um alarme. A API faz *queries* a uma base de dados para poder fazer o que for necessária para satisfazer os *requests* dos clientes.

Uma vez que a API tenha o que é necessário, manda de volta para os clientes um JSON response com o que foi pedido no *request* do cliente. O cliente ficará sempre a espera de um response uma vez que envie um *request*.

3. REST

A Representational State Transfer (REST), em português Transferência de Estado Representacional, é uma abstração da arquitetura da World Wide Web, mais precisamente, é um estilo de arquitetura que consiste num conjunto coordenado de restrições de arquiteturas aplicadas a componentes, conectores e elementos de dados dentro de um sistema de hipermédia distribuído.

O REST ignora os detalhes da implementação de componente e a sintaxe de protocolo com o objetivo de focar nos papéis dos componentes, nas restrições sobre sua interação com outros componentes e na sua interpretação de elementos de dados significantes.

Ele é frequentemente aplicado à *web services* fornecendo APIs para acesso a um serviço qualquer na web. Usa integralmente as mensagens HTTP para se comunicar através do que já é definido no protocolo sem precisar "inventar" novos protocolos específicos para aquela aplicação.

Não possui estado entre essas comunicações, ou seja, cada comunicação é independente e uniforme precisando passar toda informação necessária e deve facilitar o cache de conteúdo no cliente.

Deve ter clara definição do que faz parte do cliente e do que faz parte do servidor, permite o uso em camadas facilitando assim a escalabilidade, confiabilidade e segurança e é frequentemente criado com alguma forma de extensibilidade. **[8]**

4. HTTP *Request* e *Response*

Request* e *Response são dois importantes objetos utilizados em quase todas as linguagens de programação web existente. Diversas propriedades que envolvem cliente e server podem ser acedidas através deles.

A definição de ambos é bem simples, ***Request*** é a informação que vai para o servidor através do navegador (Input), já o ***Response*** é a informação que chega ao navegador através do servidor (Output). Com eles é possível ler diversos dados técnicos que podem ser utilizados no desenvolvimento de funcionalidades de aplicações mais complexas. **[9]**

O protocolo HTTP define uma série de Métodos de *Request* responsáveis por indicar a ação a ser executada na representação de um determinado recurso. Embora esses métodos possam ser descritos como substantivos, eles também são normalmente referenciados como verbos, os chamados *"HTTP Verbs"*.

Cada um deles implementa uma diferente função sendo que alguns recursos são comuns entre todos os verbos, por exemplo, qualquer método de *request* pode ser do tipo safe, *idempotent* ou *cacheable*. **[10]**

- **Métodos:**
  - **GET;**
  - **HEAD;**
  - **POST;**
  - **PUT;**

- **DELETE;**
- **CONNECT;**
- **OPTIONS;**
- **TRACE;**
- **PATCH.**

Os códigos de status das HTTP responses indicam se um HTTP *request* foi corretamente concluído. Os responses que utilizamos foram: **[11]**

- **[200] OK:** Significa que o *request* foi bem-sucedido. O significado do sucesso varia de acordo com o método HTTP;
- **[201] Created:** o *request* foi bem-sucedido e um novo recurso foi criado como resultado. Esta é uma típica resposta enviada após uma requisição PUT.
- **[302] Found:** Este código de resposta significa que o URI do recurso solicitado foi alterado temporariamente. Novas alterações no URI podem ser feitas no futuro. Portanto, esse mesmo URI deve ser usado pelo cliente em solicitações futuras.
- **[400] Bad Request:** Essa resposta significa que o servidor não pôde entender a solicitação devido à sintaxe inválida.
- **[404] Not Found:** O servidor não consegue encontrar o recurso solicitado. Este código de resposta é provavelmente o mais famoso devido à sua frequência de ocorrer na web.
- **[500] Internal Server Error:** O servidor encontrou uma situação que não sabe como manipular.

5. Machine Learning

O *machine learning* é um tipo de **inteligência artificial** que permite que as aplicações de software sejam bastante precisas na previsão de resultados, mesmo sem serem expressamente programadas para isso.

A aprendizagem que dá nome à expressão "*machine learning*" consiste na execução de algoritmos que criam de modo automático modelos de representação de conhecimento com base num conjunto de dados.

A ideia em que assenta esta aprendizagem é que devemos treinar as máquinas, dando-lhes acesso aos dados históricos, uma ou mais medidas de desempenho, e deixando o **algoritmo** "aprender", ou seja, ajustar de modo iterativo o modelo de representação de conhecimento de modo a que este melhore o seu desempenho.

Após este treino, o modelo tem um potencial para efetuar previsões de qualidade em situações futuras e que estejam relacionadas com padrões históricos. Esta estratégia pode mesmo ser usada para aceder à Internet e aprender de modo contínuo com a todos os dados que forem procurados.

Este é um método de análise de dados que está tão instituído no nosso dia-à-dia que praticamente não nos apercebemos da sua utilização, dada toda a nossa familiaridade com ele.

Recomendações da Amazon, pesquisas da *Web* e as traduções automáticas do serviço Google são bons exemplos pois são baseadas em algoritmos de *machine learning*.

Com o *machine learning* os computadores facilitam a nossa vida, agindo de forma rápida e subtil, apesar de muitas das vezes exigirem um grande volume de dados e processamento na sua fase de treino. **[12]**

O interesse renovado no *machine learning* deve-se ao aumento do volume e da variedade de dados disponíveis, o processamento computacional mais barato e poderoso, o armazenamento de dados acessível etc.

Tudo isto significa que é possível produzir modelos capazes de analisar dados maiores e mais complexos de forma rápida e automática, e entregar resultados igualmente mais rápidos e precisos. **[13]**

6. Linear Regression

Linear Regression é talvez um dos algoritmos mais conhecidos e bem compreendidos em estatística e *machine learning*.

O *machine learning*, está principalmente preocupado em minimizar o erro de um modelo ou em possibilitar previsões mais precisas. Na aplicação de *machine learning*, vai-se emprestar, reutilizar e roubar algoritmos de muitos campos diferentes, incluindo estatísticas e usá-los para esses fins.

Como tal, a *Linear Regression* foi desenvolvida no campo da estatística e é estudada como um modelo para entender a relação entre variáveis numéricas de entrada e saída. É um algoritmo estatístico e um algoritmo de *machine learning*. **[14]**

Linear Regression é uma equação para se estimar a condicional (valor esperado) de uma variável *y*, dados os valores de algumas outras variáveis *x*. A regressão, em geral, tem como objetivo tratar de um valor que não se consegue estimar inicialmente.

Na *Linear Regression* o nome "linear" vem do facto de se considerar que a relação da resposta às variáveis é uma função linear de alguns parâmetros. Os modelos de regressão que não são uma função linear dos parâmetros chamam-se modelos de regressão não-linear.

Foi uma das primeiras formas de análise *regressiva* a ser estudada rigorosamente, e usada extensamente em aplicações práticas. Isto acontece porque modelos que dependem de forma linear dos seus parâmetros desconhecidos, são mais fáceis de ajustar que os modelos não-lineares aos seus parâmetros, e porque as propriedades estatísticas dos estimadores resultantes são fáceis de determinar.

Modelos de regressão linear são frequentemente ajustados usando a abordagem dos mínimos quadrados, mas que também pode ser montada de outras maneiras, tal como minimizando a "falta de ajuste" numa outra norma, ou através da minimização de uma penalização da versão dos mínimos quadrados.

Por outro lado, a abordagem de mínimos quadrados pode ser utilizada para ajustar modelos que não são modelos lineares. Assim, embora os termos "mínimos quadrados" e "modelo linear" estejam intimamente ligados, não são sinônimos.

Para se estimar o valor esperado, usa-se de uma equação, que determina a relação entre ambas as variáveis. **[15]**

$$Y_i = \alpha + \beta X_i + \epsilon_i$$

Em que:

- Y_i - Variável explicada, o valor que se quer atingir;
- α - É uma constante, que representa a intercetação da reta com o eixo vertical;
- β - É outra constante, que representa o declive da reta;
- X_i - Variável explicativa, representa o fator explicativo na equação;
- ϵ_i - Variável que inclui todos os fatores residuais mais os possíveis erros de medição. O seu comportamento é aleatório, devido à natureza dos fatores que encerra.

Cálculo dos fatores alfa e beta:

$$\hat{\alpha} = \frac{\sum X^2 \sum Y - \sum (XY) \sum X}{n \sum X^2 - (\sum X)^2}$$

$$\hat{\beta} = \frac{n \sum (XY) - \sum X \sum Y}{n \sum X^2 - (\sum X)^2}$$

Definindo $\overline{X} = \frac{\sum X}{n}$ e $\overline{Y} = \frac{\sum Y}{n}$, temos que $\hat{\alpha}$ e $\hat{\beta}$ se relacionam por:

$$\hat{\alpha} = \overline{Y} - \hat{\beta}\,\overline{X}$$

Estas fórmulas podem ser desenvolvidas a partir da definição de mínimos quadrados. O objetivo é determinar alfa e beta de forma que a soma dos quadrados dos erros seja mínima, ou seja, devemos minimizar

$$\sum (Y_i - \beta X_i - \alpha)^2$$

Desenvolvendo este quadrado e eliminando os termos constantes obtém-se:

$$\beta^2 \sum X^2 + n\alpha^2 - 2\beta \sum (XY) - 2\alpha \sum Y + 2\alpha\beta \sum X$$

A partir desse ponto, pode-se resolver usando-se cálculo ou através de uma transformação de coordenadas:

$$\alpha = \alpha_1 - \frac{\sum X}{n}\beta \qquad \text{ou} \qquad \alpha = \alpha_1 - \beta\overline{X}$$

Transformando a expressão a ser minimizada em:

$$\beta^2 \sum X^2 + n\alpha_1^2 - 2\alpha_1\beta\sum X + \frac{(\sum X)^2}{n}\beta^2 - 2\beta\sum(XY) - 2\alpha_1\sum Y + 2\overline{X}\sum Y\beta + 2\alpha_1\beta\sum X - 2\frac{(\sum X)^2}{n}\beta^2$$

ou

$$\beta^2 \sum X^2 + n\alpha_1^2 - \frac{(\sum X)^2}{n}\beta^2 - 2\beta\sum(XY) - 2\alpha_1\sum Y + 2\overline{X}\sum Y\beta$$

Esta expressão se separa na soma de duas expressões quadráticas independentes, que podem ser minimizadas usando matemática elementar:

$$n\alpha_1^2 - 2\alpha_1\sum Y$$

$$\beta^2\sum X^2 - \frac{(\sum X)^2}{n}\beta^2 - 2\beta\sum(XY) + 2\frac{\sum X\sum Y}{n}\beta$$

Cujos valores minimizadores são:

$$\alpha_1 = \frac{\sum Y}{n}$$

$$\alpha = \overline{Y} - \overline{X}\beta$$

$$\beta = \frac{n\sum(XY) - \sum X\sum Y}{n\sum X^2 - (\sum X)^2}$$

7. Persistência de modelos de machine learning

Depois de treinar um modelo scikit-learn **[16]**, é desejável ter uma maneira de persistir o modelo para uso futuro sem ter que reciclar. Para persistir um modelo normalmente usa-se o pickle.

No nosso em vez pickle optamos por usar o joblib (joblib.dump & joblib.load), que é mais eficiente em objetos que transportam grandes matrizes internamente.

O joblib tem alguns problemas relacionados à manutenção e segurança. Por causa disso nunca se deve usar em dados não confiáveis, pois isso pode levar a códigos maliciosos sendo executados durante o carregamento.

Embora os modelos salvos usando uma versão do scikit-learn possam ser carregados em outras versões, isso é totalmente sem suporte e desaconselhável. Também deve ser mantido em mente que as operações executadas em tais dados podem dar resultados diferentes e inesperados.

Para reconstruir um modelo semelhante com versões futuras do scikit-learn, meta dados adicionais devem ser guardados ao longo do modelo em pickled: **[17]**

- Os dados de treino;
- O código-fonte Python usado para gerar o modelo;
- As versões do scikit-learn e suas dependências;
- A pontuação de validação cruzada obtida nos dados de treino.

8. Entity Framework

O ADO.NET Entity Framework é uma das principais ferramentas de persistência presentes na plataforma .NET, sendo parte integrante do pacote de tecnologias ADO.NET.

Proporciona soluções para minimizar o problema de impedância, abstraindo do *developer* vários detalhes das bases de dados relacionais. Além disso, fornece uma série de recursos que aumentam muito a produtividade no desenvolvimento de aplicações persistentes. **[18]**

9. Connection String

Cada provedor de dados do .NET Framework possui um objeto Connection que herda de DbConnection, bem como uma propriedade ConnectionString específica do provedor.

A sintaxe da sequência de conexão específica para cada provedor é documentada na sua propriedade ConnectionString. Dos quatro provedores de dados incluídos no .NET Frameworko que é usado por nós é o System.Data.SqlClient que fornece acesso a dados para o Microsoft SQL Server. **[19]**

10. LINQ Queries

Uma consulta é uma expressão que recupera dados de um *data source*. As consultas geralmente são expressas numa linguagem de consulta especializada.

Diferentes linguagens foram desenvolvidas ao longo do tempo para os vários tipos de *data sources*, por exemplo, SQL para base de dados relacionais e XQuery para XML.

Portanto, os developers tiveram que aprender uma nova linguagem de consulta para cada tipo de *data source* ou formato de dados que devem suportar. O LINQ simplifica essa situação oferecendo um modelo consistente para trabalhar com dados em vários tipos de origens e formatos de dados.

Em uma consulta LINQ, está-se sempre a trabalhar com objetos. Usa-se os mesmos padrões básicos de codificação para consultar e transformar dados em documentos XML, bases de dados SQL, conjuntos de dados ADO.NET, coleções .NET e qualquer outro formato para o qual provedor LINQ esteja disponível. **[20]**

Todas as operações de consulta do LINQ consistem em três ações distintas:

- Obter o *data source*;
- Criar a consulta;
- Executar a consulta.

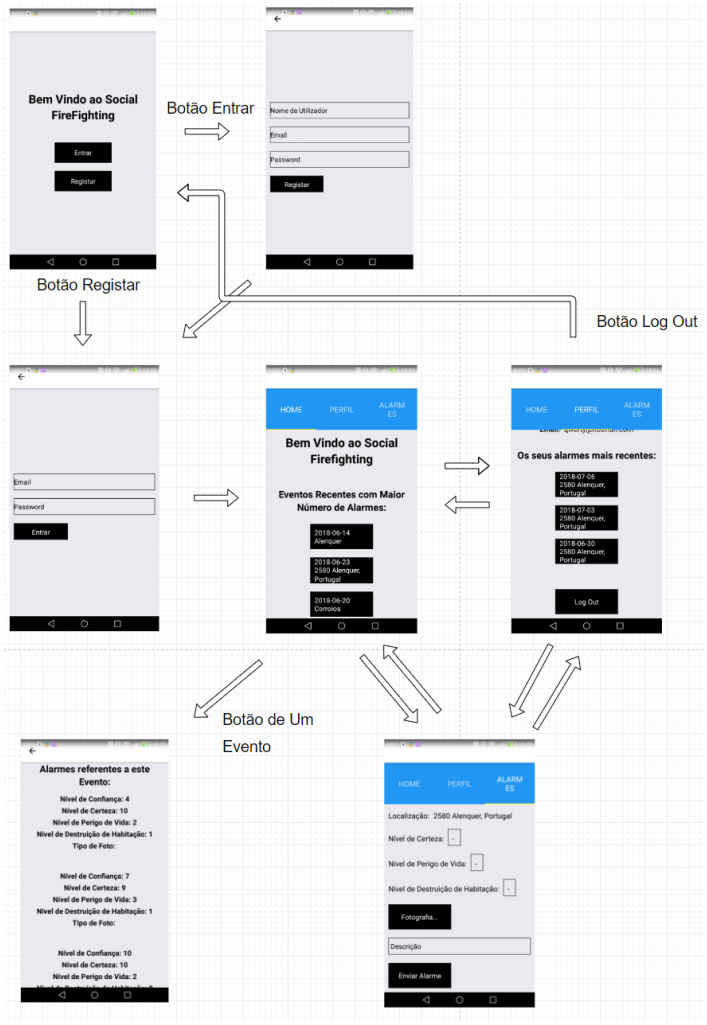
Método/Desenvolvimento

O nosso projeto passou pelo desenvolvimento de um *front-end*, para a apresentação de interfaces aos utilizadores, e de um *back-end*, para armazenamento e tratamento de dados. Neste desenvolvemos não só o *front-end* para apresentação aos cidadãos, o objetivo principal, mas também um de apresentação aos bombeiros, um objetivo extra, desenvolvendo para tal um correspondente *back-end* que permita o seu bom e correto funcionamento.

Front-end

O desenvolvimento do *front-end* passou pela criação de dois recursos: uma **aplicação móvel** para apresentação de uma interface para os utilizadores (os cidadãos), conectada a uma API, de modo a esta aplicação conseguir comunicar e fazer pedidos para apresentação de dados da base de dados (o nosso *back-end*); uma **aplicação *web*** para apresentação de uma interface para os utilizadores (os bombeiros), também conectada á mesma API, de modo a esta aplicação poder comunicar e fazer os pedidos para apresentação dos dados da base de dados. Assim, geralmente, o *front-end* são as aplicações que servem para apresentar interfaces, conectadas ao *back-end* através de uma API.

Front-end – App



Para o desenvolvimento da aplicação utilizámos a linguagem React Native. O React Native é um framework baseado no já aclamado React, desenvolvido pela equipa do Facebook, que possibilita o desenvolvimento de aplicações mobile, tanto para Android**,** como para iOS, utilizando apenas Javascript.

Uma vez que o nosso objetivo era realizar uma aplicação que funciona tanto em iOS como android, escolhemos o React Native graças a sua excelente performance, velocidade de carregamento e excelente integração com funcionalidades de telemóvel como a câmara que seria um elemento fundamental na nossa aplicação. **[21]**

Para podermos começar a programar React Native, começamos por instalar o Node.js que é basicamente uma plataforma para construir aplicações web escaláveis de alta performance usando JavaScript.

Uma vez com o Node.js instalado, decidimos usar o Expo para começar a programar. O Expo permite que se crie aplicações de *cross-platform* apenas usando JavaScript, permito o uso de qualquer editor de texto para escrever componentes de React Native sem ser necessário o Xcode ou o Android Studio. **[22]**

Para a utilização do Expo é necessário utilizar a linha de comandos do computador para poder instalá-lo e criar uma aplicação. Uma vez que o Node.js esteja instalado, é necessário na linha de comando usar o comando *"npm install exp -global"* para instalar o expo na respetiva máquina e após a instalação criar um projeto com o comando *"exp init nome-do-projeto"* sendo que nome-do-projeto será o nome dado ao projeto.

Com o Expo instalado e com o projeto criado, para correr o projeto basta com o projeto selecionado na linha de comandos usar o comando "*exp start"* e o programa será corrido em qualquer telemóvel que tenho o expo instalado e na mesma rede de onde o projeto foi iniciado.

Na programação em si, utilizámos o Notepad++ e usamos os nossos telemóveis e emulador de android para poder correr a aplicação. Na aplicação programamos 6 ficheiros diferentes: a *app*, uma página para escolha de registo ou *login*, uma página de *login*, uma página de registo, a *home page* e uma página com a informação dos eventos.

A primeira página programada trata-se da app.js, a *app* é o começo, é a primeira página a correr quando a aplicação é iniciada. O único código que a página contém é a implementação do *stack navigator*.

Stack navigator é um componete do React Native que permite aos utilizadores navegar de página em página sendo que cada página é chamada de *stack* e uma vez se se salta de uma *stack* para outra existe a possibilidade de voltar à *stack* anterior.

Nós utilizados o *Stack navigator* na nossa aplicação para o funcionamento do fluxo da aplicação em si, ou seja, para permitir saltar da página seleção de registo ou *login* para a página respetiva, saltar da página de registo para a página seleção de registo ou *login*, saltar da página de *login* para o *home* e saltar do *home* para a página de informação de um evento quando este é selecionado na *home page*.

Este código está implementado na app.js e sendo que o fluxo da *stack* começa na página de seleção de registo ou *login* a app.js faz com que a *stack* comesse por lá.

O *Stack navigator* não vem predefinida no React Native, sendo que por sua vez exige um *import* que é feito na linha de comandos com o seguinte comando: *"npm install --save react-*

navigation". Utilizamos as propriedades *"screen"* para atribuir à *stack* o nome do ficheiro e a propriedade *"navigationOptions"* que por sua vez possui a opção *"headerLeft: null"*, que utilizamos para não permitir o retorno à stack anterior na página do *home* uma vez que a única maneira de sair da página do *home* será dando *logout*, e para não permitir voltar à *stack* anterior na página de seleção de registo ou login uma vez que é possível aceder a essa página vindo de um registo.

A página introLogIn é o nosso ficheiro que permite ao utilizador escolher se quer entrar com uma conta que já esta criada ou se quer se registar na aplicação. É a primeira página a ser acedida uma vez que a aplicação é executada.

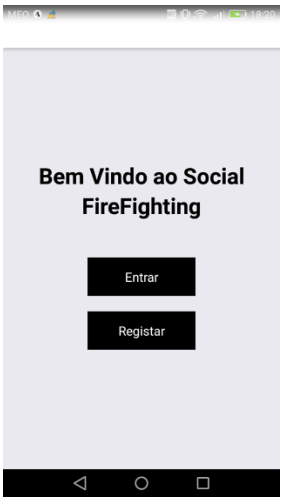
A página de seleção de registo ou *login* contém apenas o uso de uma mensagem de boas vindas ao utilizador e dois botões, um para saltar para a página de registo e outro para saltar para a página de *login*.

Para o design da página utilizamos o componente <*view*> que é o componente mais importante para construir uma interface de utilizador, o *View* é um *container* que suporta layout com *flexbox*, estilo *handling touches* e acessibilidade.

Não só o *View* desta página, mas todos os *Views* utilizados na aplicação usam o *flex* a 1, ou seja, usa o *View* como um todo sem divisão e o *justifyContent* e o *alignItems* ao centro para ter todo o conteúdo da aplicação centrados no meio da página.

Dentro dessa *View* usamos um <*Text*> (componente utilizado para escrever texto) para escrever a mensagem de boas vindas à aplicação e utilizámos 2 <*TouchableOpacity*> que é um *wrapper* para que as visualizações respondam adequadamente aos toques, ou seja, são basicamente botões, que utilizamos para saltar para a página de registo ou para a página de *login*.

No *TouchableOpacity* usamos a propriedade *onPress* para saltar para as páginas respetivas e o *style* para atribuir um estilo ao *TouchableOpacity* que ao longo da aplicação é sempre usado.



Caso o utilizador decida clicar no botão de entrar na página de seleção de *login* ou entrar, será reencaminhado para a página de entrar. Esta página é bastante simples em termos de design, contém 2 <*TextInput*> que é simplesmente uma caixa de texto onde o utilizador pode colocar informação, onde usamos uma para o utilizador introduzir o email e outra para o utilizador introduzir a password.

Estes *TextInputs* usam a propriedade *placeholder* para poder aparecer por defeito “email” e “password” nas respetivas caixas de texto, a propriedade *onChangeText* para quando o utilizador escrever permitir guardar a informação escrita numa variável e no *TextInput* da password usa a propriedade *“secureTextEntry={true}”* para esconder a informação escrita na password. A página contém também um botão *TouchableOpacity* para fazer o utilizador autenticar-se, ou seja, se a o email e password estiverem corretos a aplicação salta para a stack do home.

A página do entrar, ou seja o Entrar.js possui uma função *registerForPushNotificationsAsync* que tem como objetivo a gerar um *token* a partir do *Notifications.getExpoPushToken* e enviá-lo para a página *home* uma vez que o utilizador dê login.

Isto é utilizado para o sistema de notificações, logo existe também a necessidade de usar o *Permissions.NOTIFICATIONS* e garantir que o utilizador permite que a aplicação tenha permissão, sendo que verificamos se a permissão foi concedida antes de gerar o *token*.

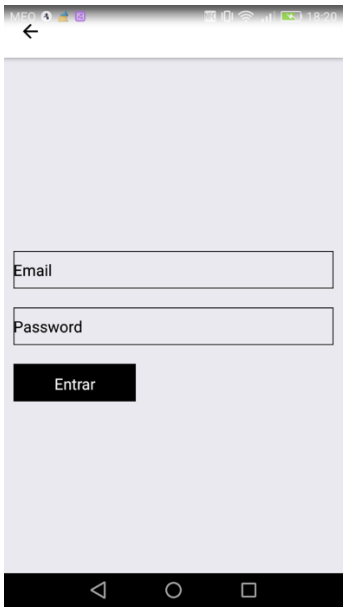
Outra função usada na página entrar é o *componentDidMount. componentDidMount* é uma função que é executada assim que o ficheiro é corrido, e no entrar, nós usamos esta função para chamar a função *registerForPushNotificationsAsync* com o objetivo de gerar um *token* assim que a o ficheiro Entrar.js seja executado.

Por fim, na página do entrar temos uma função chamada UserLoginFunction e é nesta função que a funcionalidade de login acontece, esta função é chamada pelo botão login, ou seja, assim que o utilizador clicar no botão esta função é executada.

A função contém um *fetch* do método *post* para fazer um *request* à API, para o body do *fetch* é enviado o valor da variável email e da variável password, que correspondem à informação que está nos *TextInputs* no momento em que o utilizador clica no botão entrar.

No fim da função, a partir do response existe a aplicação verifica se o status é igual a 200, se sim, existe sucesso e a aplicação monstra um alerta que diz que houve sucesso no *login* e salta para a página do *home* levando o id, o email e o *token*.

Caso o status seja diferente de 200 significa que aquela combinação de email e password não existe, logo não possibilita o *login* e mostra um alerta que diz que a combinação de email e password estão incorretos.



Na página de registar, o design da página é muito semelhante ao do entrar, só que neste caso para além do *TextInput* do email e da password existe também o *TextInput* do nome e sendo que este formulário serve para o registo o botão *TouchableOpacity* possui *um Text* que diz registar em vez de entrar.

Tendo isto em conta, o design do registar consiste numa *View* que possui 3 *TextInputs*, um para o utilizador introduzir o email, outro para o utilizador introduzir a password e outro para o utilizador introduzir o nome.

Estes *TextInputs* usam a propriedade *placeholder* para poder aparecer por defeito "email", "password" e "nome" nas respetivas caixas de texto, a propriedade *onChangeText* para quando o utilizador escrever permitir guardar a informação escrita numa variável e no *TextInput* da password usa a propriedade *"secureTextEntry={true}"* para esconder a informação escrita na password.

A página contém também um botão *TouchableOpacity* para fazer o utilizador registar-se, ou seja, se a o nome, o email e a password estiverem num formato correto a conta será registada e a aplicação irá saltar de volta para a *stack* da seleção de registo ou *login*.

Para além do design, o ficheiro registar.js possui uma função para fazer com que o funcionamento do registo ocorra.

Essa função chama-se *UserRegistrationFunction,* esta função é chamada pelo botão de registar, ou seja, assim que o utilizador clicar no botão esta função é executada.

A função contém um *fetch* do método *post* para fazer um *request* à API. No body do *fetch* vai o nome, o email e a password que correspondem ao valor das respetivas TextInputs no momento em que o utilizador clica no botão do registo. Para além dessas 3, vai também o tipo_user, que envia sempre "Cidadão" por defeito, pois quem criar um registo a partir da aplicação será sempre um cidadão.

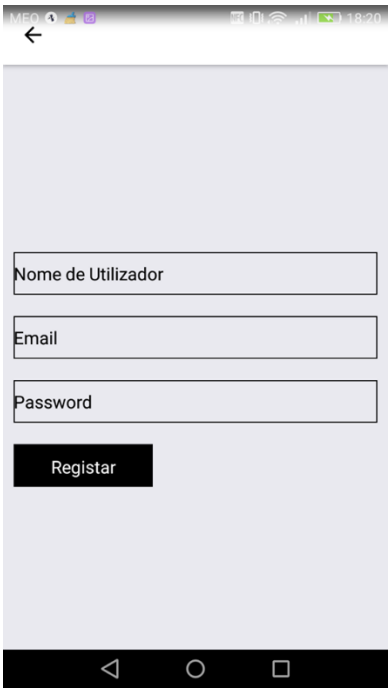
Por fim no body vai também o ranking de confiança que significa o valor de confiança de 0-10 desse cidadão. Esse valor é sempre por defeito 5 que é o nosso valor neutro, ou seja, o valor que é atribuído a um cidadão assim que esta cria uma conta.

No Fim da função existem a aplicação vai confirmar se o nome o email e a password introduzidos pelo utilizador estão num formato valido.

Primeiro confirma-se que tanto o campo do nome, do email e da password não estão vazios, pois são todos obrigatórios. Caso esta ação se verifique então o código passa para a segunda *validação* que serve para confirmar que o campo de email está num formato de email valido.

Se ambas as verificação se confirmem então o código vai confirmar que o email introduzido ainda não está em uso, para tal usamos o status 302, que serve para isso mesmo. No fim o programa vai verificar se o status é igual a 201, ou seja se existiu sucesso no registo.

Em cada uma das verificações do *UserRegistrationFunction* existe um alerta que mostra ao utilizador uma mensagem do porquê de não ter conseguido efetuar o registo. Se houver sucesso, a aplicação irá mostrar uma mensagem de sucesso e saltará para a *stack* da página de seleção de login ou registo.



Uma vez que um utilizador tenha feito login, a aplicação vai saltar para a stack do home. No home é onde está a maior parte do código e é aqui que os utilizadores podem ver os últimos eventos, visitar o seu perfil e enviar alarmes.

Para fazer esta divisão utilizamos um *import* do React Native chamado de *TabNavigator*. *TabNavigator* é possivelmente, o estilo mais comum de navegação para aplicações mobile, onde a navegação é baseada em tabs. As *tabs* podem ser usadas na parte superior da página ou na parta inferior da página, no nosso caso para a versão iOS as *tabs* estão na parte inferior e no Android na parte superior.

Na nossa aplicação utilizamos o *TabNavigator* para criar 3 *tabs*: *home*, que é basicamente a página que da as boas vindas ao utilizador que deu login e mostra os últimos 5 eventos que ocorreram, o perfil, que é onde o utilizador vê a sua informação e os seus últimos 3 alarmes enviados e por fim a *tab* do alarme que se trata de um formulário para o utilizador preencher e enviar um alarme.

Cada uma das *tabs* está associada a uma classe, onde dentro de cada uma dessas classes está o código referente a essa *tab*. Dentro do *TabNavigator* utilizamos a propriedade *tabBarOptions* para poder adicionar opções a nossa *TabNavigator*.

A primeira classe é a *home* que corresponde a primeira *tab* que é acedida assim que o utilizador faz login na aplicação. Na parte do design esta aplicação possui a *tag* <*ScrollView*> que serve para o utilizador poder fazer *scroll* na *tab* em especiífico. Possui também uma *View* que serve de controlo para todo o design.

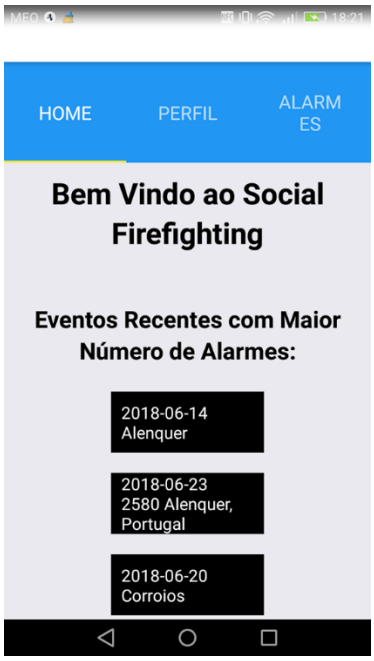
Dentro dessa *View* existe 2 *Texts*, um deles mostra a mensagem de boas vindas e o outro mais a baixo funciona como um título para mostrar os últimos 5 eventos.

Por baixo dessa mensagem existem 5 botões *TouchableOpacity*, cada um deles referente a um dos 5 eventos. Ao clicar num deles a aplicação irá saltar para a *stack* dos eventos, onde mostrará a informação referente a esse evento.

Cada um desses botões tem lá escrito a data em que começou e a localidade onde se encontra. Possui a propriedade *onPress* que usa o *this.props.navigation.navigate* para saltar para a *stack* do eventos e levar o valor do id desse evento.

O código referente aos 5 botões é feito apenas um botão que é corrido 5 vezes que corresponde ao número de eventos dentro de um array criado na classe, que por sua vez é sempre 5, pois o nosso objetivo é mostrar sempre os últimos 5 eventos.

Para colocar no array a informação dos eventos utilizamos a função *componentDidMount* para fazer um *fetch* do tipo *GET* para receber os eventos, sendo que graças ao *componentDidMount*, o array vai receber os valores assim que o *home* for iniciado.



A Segunda classe é a classe do perfil que corresponde a a tab do perfil do utilizador que fez login. Na parte do design esta aplicação possui a *tag* <*ScrollView*> tal e qual como na tab do *home* tem como objetivo poder dar *scroll* na *tab*.

Possui também uma *View* que serve de controlo para todo o design. Dentro dessa *View* existe um *Text* que funciona como título da *tab* que diz "informação de utilizador" e duas outras *Texts* que servem para mostrar o nome e o email do utilizador que fez *login*.

À frente desses dois *Texts* existem outros dois que usam o *this.props.navigation*.state.*params* para ir buscar o nome do utilizador e o email que foram enviados pelo formulário de *login* e assim poder mostrar no perfil a informação referente ao utilizador em questão.

Muito à semelhança da *tab* do *home*, por baixo da informação do utilizador existe um *Text* que introduz os últimos 3 alarmes feitos por o utilizador, ou seja, o código referente aos 3 alarmes é feito apenas um *Text* que é corrido 3 vezes que corresponde ao número de alarmes dentro de um array criado na classe, que igualmente aos eventos tem um número fixo, neste caso 3 pois o nosso objetivo é mostrar sempre os últimos 3 alarmes feitos pelo utilizador. A informação de cada alarme corresponde à data e ao local de quando e onde foi emitido esse alarme.

Igualmente ao *home*, para colocar no array a informação dos alarmes utilizamos a função *componentDidMount* para fazer um *fetch* do tipo *GET* para receber os últimos 3 alarmes feitos pelo utilizador autenticado, sendo que graças ao *componentDidMount*, o array vai receber os valores assim que o *home.js* for iniciado.

Por fim, existe um *TouchableOpacity* que tem como texto "Logout" e serve assim para o utilizador sair da sua conta. Ao clicar neste botão o utilizador saíra do *home* e voltará a página de início para escolher se quer dar *login* ou se quer se registar.

A terceira e última classe do *TabNavigator* é a classe alarmes que é a classe que deu mais trabalho, pois foi onde usamos uma grande quantidade de *imports* e consequentemente fez com que fosse o bloco de código mais difícil de produzir.

Esta *tab* consiste basicamente num formulário que o utilizador pode preencher para poder enviar um alarme. Em termos de design do formulário em geral existe um *ScrollView* para poder fazer *scroll* na página e de um *View* que engloba todo o design.

Dentro da *View* principal do alarme, existiu necessidade de criar várias *Views* dentro da *View* principal, onde cada uma dela refere-se a um espaço necessário para poder enviar o alarme. O primeiro *View* em termos de propriedades possui 1 de *felx* e *row* na *flexDirection* para poder criar o *view* de forma horizontal.

Este primeiro *View* é referente à *View* que mostra a localização do utilizador. Para a localização não é necessário o utilizador introduzir a sua localização, pois a aplicação automaticamente vai buscar a localização do utilizador a partir da sua longitude e latitude.

Dentro dessa *View* existem 2 *Texts*, um deles feito manualmente que diz "localização:" e o outro que a frente da localização mostra a variável "local" que possui a localização do utilizador.

O segundo *View*, que vem seguido da localização é o nível de certeza. Nível de certeza é onde o utilizador irá introduzir um número de 1 a 10 que indicará a sua certeza em relação ao

incendio. Este *View* possui as mesmas propriedades que o *View* anterior, mas agora em vez de 2 *Texts* possui um *Text* e um *ModalDropdown*.

O *Text* é exatamente igual ao que diz "localização" no *View* anterior, mas neste caso diz "Nível de Certeza" e à frente desse *Text* está então o *ModalDropDown*. *ModalDropDown* é um import que foi feito para poder ser utilizado, e para tal usamos "npm i react-native-modal-dropdown -save".

ModalDropdown é um *dropdown*/*picker*/*selector* para o React Native. Usa javascript puro, é compatível tanto com iOS como Android, usa uma posição automática (Não será coberto ou cortado pela borda da tela) e é altamente personalizável.

Usamos então o *ModalDropdown* para fazer um *dropdown* com 10 valores (1-10) para o utilizador poder selecionar o seu nível de certeza. As propriedades usadas no *ModalDropdown* são o *textStyle* para definir um estilo do texto que aparece dentro do modal, *dropdownStyle* para definir um estilo para o *dropdown* em si.

Usamos a propriedade *defaultValue* a "-" que será o que aparece até o utilizador definir um valor, usamos a propriedade "*options*" onde colocamos lá então os 10 valores e por fim a propriedade *onSelect* que é a propriedade que guarda o valor selecionado numa variável global.

O terceiro *View* é o *View* referente ao nível de perigo de vida que é onde o utilizador irá colocar um valor de 1 a 10 de quanto acha que existe perigo de vida nesse alarme. Igualmente ao *View* anterior este tem as mesmas propriedades, um *Text* e um *ModalDropdown*.

As propriedades deste *ModalDropdown* deste *View* são as mesmas do *ModalDropdown* do nível de certeza, usamos o mesmo *textStyle* e o mesmo *dropdownStyle*, usamos a propriedade *defaultValue* a "-" e a mesma propriedade "*options*" onde colocar os 10 valores e por fim a propriedade *onSelect* guarda igualmente o valor selecionado numa variável global.

O quarto *View* foi igualmente feito em comparação com o nível de certeza e com o nível de perigo de vida. Neste caso temos o nível de destruição de habitação, que é onde o utilizador usa a classificação de 1 a 10 para classificar o perigo de haver habitação danificada devido ao incêndio.

Este *View* contém assim também um *Text* que diz "Nível de Destruição de Habitação:" e mais frente outro *ModalDropdown* com os valores de 1-10 para o utilizador poder escolher. As propriedades desse *ModalDropdown* são as mesmas que os dois *ModaDropDowns* anteriores, mas neste caso a propriedade *onSelect* guarda o valor do nível de destruição de habitação numa outra variável diferente.

Depois destes 4 *Views* existe um TouchableOpacity, ou seja, um botão que permite ao utilizador escolher uma foto das suas fotos do telemóvel para poder ser enviada essa fotografia no alarme. Para tal esse botão usa o *style* para usar o estilo que é usado em todos os *TouchableOpacity* da aplicação e usa o *onPress* para chamar a função EscImagem que fará com que a foto escolhida pelo utilizador seja colocada na variável image.

Por Baixo desse botão existe um espaço em que usará a tag <Image> para mostrar a foto que foi escolhida pelo utilizador.

A seguir à seleção de foto existe um *TextInput* para o utilizador poder escrever uma descrição sobre o incêndio, sendo que este campo não é obrigatório. Este *TexInput* possui como propriedades o *placeholder* para poder aparecer descrição na caixa de texto até o utilizador escrever

alguma descrição, o *placeholderTextColor* para definir a cor do texto dentro da caixa de texto a preto e o *onChangeText* para guardar o que for escrito pelo utilizador numa variável descrição.

No fim do formulário existe então um botão que serve para enviar o alarme. Simplesmente este *TouchableOpacity* usa o *style* geral da aplicação e usa o *onPress* para chamar a função AlarmeFunction que irá tratar do envio do alarme para a API.

Em relação às funções usadas nesta classe foram várias, a principal foi no *componentDidMount* onde fizemos várias funções dentro dele. Tendo isto em conta a primeira função que temos é o *componenteDidMount* que é a função que irá ser executada assim que a aplicação chegar a esta *tab*.

Dentro do *componentDidMount*, temos a função requestLocation. Houve a necessidade de fazer esta função pois o código que o programa possuía para poder ter permissões à localização do utilizador só funcionava em iOS, quando fomos testar não funcionava em Android, havendo então assim necessidade de implementar esta função.

Esta função requestLocation é uma *async functions* e usa o PermissionsAndroid.PERMISSIONS.ACCESS_FINE_LOCATION que é uma funcionalidade de um *import* do React Native para poder ter acesso à localização do utilizador. Para tal existe a necessidade de usar o "npm install --save react-native-permissions" para poder usar esta funcionalidade e então assim a primeira vez que a aplicação for corrida num novo telemóvel irá pedir por permissão ao utilizador se pode aceder a sua localização e assim permitir que a aplicação use a localização para fazer o envio do alarme.

Depois da função requestLocation ainda dentro da *componentDidMount*, usamos mais um *import*, desta vez o *geolocation* que serve para aceder a localização geográfica do telemóvel. Para fazer o import usamos "npm install --save react-native-geocoding".

Uma vez com o *import* instalado, usamos navigator.geolocation.getCurrentPosition e essa *position* fornece-nos a latitude e a longitude do telemóvel em especifico, graças a isso usamos esta função para poder guardar o valor da latitude e da longitude do utilizador numa variável.

Depois do navigator.geolocation.getCurrentPosition ainda dentro do *componentDidMount* fazemos um *fetch* do tipo POST para enviar para a API o nome da localização do utilizador para no futuro caso exista um incêndio na zona desse utilizador ele poder ser notificado.

Dentro desse *fetch*, quando obtemos o response verificamos se ele é igual a 200, ou seja, se existiu sucesso no envio. Se sim é feito por sua vez outro *fetch* mas desta vez em vez de ser feito à nossa API é feito a uma outra API do expo (**https://exp.host/--/api/v2/push/send**) que é também do tipo POST e onde enviamos o *token*, que foi gerado na página do login, enviamos um titulo e um corpo da mensagem de notificação e assim o utilizador poderá ser notificado caso exista um incêndio na sua zona.

Já fora do *componentDidMount* temos a função getLocalização que é uma função que a partir da latitude e da longitude geram o nome da localização de onde essa latitude e essa longitude pertencem.

Para realizar o getLocalização foi necessário usar o *import* do *geocode* (npm install --save react-native-geocoder) e depois fornecer uma chave que nos foi gerada pela google para poder usar a funcionalidade do *geocode*. Assim podemos guardar numa variável localização o nome da localização do utilizador.

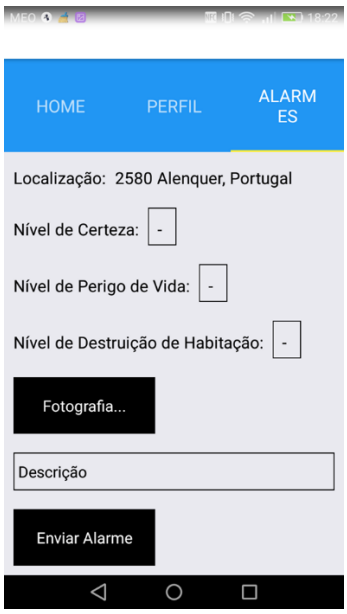
Temos depois a função chamada AlarmeFunction e é nesta função que a funcionalidade de enviar o alarme acontece, esta função é chamada pelo botão enviar alarme, ou seja, assim que o utilizador clicar no botão esta função é executada.

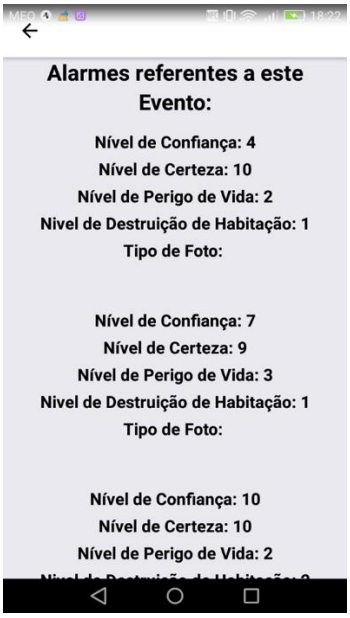
A função contém um *fetch* do método *post* para fazer um *request* à API, para o body do *fetch* é enviado o valor do nível de certeza, do nível de perigo de vida e do nível de destruição de habitação que estão no ModalDropdown. Envia também a foto que tiver sido carregada pelo utilizador, a localização que foi descoberta pela função getLocalização, a latitude e a longitude carregados pelo navigator.geolocation.getCurrentPosition, envia também o id do utilizador que veio do formulário do login, envia a descrição que tenha sido escrita no respetivo TextInput e por fim enviar a data do dia do alarme que foi anteriormente feito por nós e carregado para uma variável.

No fim da função, a partir do response existe um *if* que verifica se o status é igual a 201, se sim, existe sucesso e a aplicação monstra um alerta que diz que houve sucesso no envio do alarme. Antes de confirmar se o status é igual a 201, a aplicação vai verificar se o nível de certeza, o nível de perigo de vida e o nível de destruição de habitação, que são campos obrigatórios, estão todos preenchidos.

Por fim a última função é a função EscImagem que é a função que trata de trabalhar a imagem. Esta função é chamada quando o utilizador clica no botão de enviar fotografia que está presente no formulário de envio de alarmes.

Esta função EscImagem serve para usar o comando await e esperar que o utilizador dê permissões de acesso à camara e à localização. Uma vez com essas permissões garantidas, a função pega na imagem transforma num formato de 4:3 e guarda na variável image.





Front-end – Site

Para além da aplicação, o grupo desenvolveu também um site que é utilizado apenas pelos bombeiros. O site serve para os bombeiros poderem se autenticar e classificar eventos que existam como sendo realmente um incendio ou não.

Ao contrário da aplicação, que foi feita para telemóvel, o site é acedido através de um browser. Esta parte era um extra da aplicação que o grupo optou por fazer.

Para a realização do site utilizámos HTML, que é o formato padrão para criação de páginas online e aplicações web, usámos CSS para definir a aparência das páginas do nosso site e usámos também javascript para controlar o HTML e o CSS para manipular os comportamentos na página.

O site em si tem bastantes semelhanças com a aplicação, usamos uma página *home* que serve para o bombeiro decidir se quer criar conta ou entrar com uma conta já existente igualmente à página que a aplicação também tem.

Se o utilizador decidir dar login, o site vai para uma página para o utilizador dar login e uma vez que o login seja feito o site salta para a página com a lista dos eventos não classificados. Se o utilizador decidir se registar, o site vai para uma página para o utilizador se registar e uma vez que o utilizador se registe o site salta de volta para o *home*.

Dentro da página dos eventos, a página contém uma lista com todos os eventos não classificados e com um botão para o bombeiro poder dar *logout*. Existe também a possibilidade de o bombeiro selecionar um dos eventos e salta para a página do evento em questão.

Por fim a página do evento contém toda a informação desse evento e dois botões um que diz "incêndio" e o outro que diz "não incêndio" e assim o bombeiro de acordo com qual dos botões escolhe classifica o evento.

Sendo que estamos a trabalhar em HTML, para fazer a página home, dentro da tag head usámos a tag style para poder definir estilo para o body em geral e para os botões de login e registar.

Dentro do style do body temos width: 760px, margin:100 auto e o background-color: #999966. No style do botão definimos o background-color: #1f1f14, position: relative, border: none, font-size: 28px, color: #FFFFFF, padding: 11px, width: 150px, text-align: center e cursor: pointer.

Dentro da *tag body* temos então todo o conteúdo dentro da *tag center* para poder termos tudo centrado no meio da página e dentro da *tag* h1 temos o título da página que diz "Bem-Vindo ao Social Firefighting".

Temos por fim dois botões, um que diz login e outro que diz registar e ambos têm a propriedade *onClick* para saltarem para a página respetiva.

Depois da página do home temos em tão a página do login que, dentro da tag head usa também a tag style para definir o estilo da página. Usamos o mesmo estilo para o body e para os botões, mas nesta página temos duas caixas de texto para as quais foi necessário definir também um estilo que optamos pelo seguinte: font-size: 20px, padding: 7px, width: 270px, text-align: center.

Dentro da *tag body* temos então todo o conteúdo dentro da *tag center* para poder termos tudo centrado no meio da página e dentro da *tag* h1 temos o título da página que diz "Login".

Sendo que esta página consiste num formulário, decidimos usar a tag form para criá-lo e assim, dentro desse form usamos duas tags h2, uma para escrerver "Email" e outra para escrever "Password" onde por sua vez, usamos duas caixas de texto à frente desses textos para o utilizador introduzir o email e a password nos sítios respetivos.

Por baixo dessas caixas existe então um input do tipo botão que diz login, ou seja, assim que o utilizador clicar nesse botão o programa vai chamar a nossa função login, isto graças a propriedade onClick.

Dentro da *tag script* foi onde fizemos a nossa classe login. No início da função existem dois *ifs*, um que que verifica se existe formação nas caixas de texto e caso não exista mostra um alerta. E o outro *if* verifica se o email está num formato válido.

A função contém um *fetch* do método *post* para fazer um *request* à API, para o body do *fetch* é enviado o valor da variável email e da variável password, que correspondem à informação que está nas caixas de texto no momento em que o utilizador clica no botão login.

No fim da função, a partir do response existe um *if* que verifica se o status é igual a 200, se sim, existe sucesso e a aplicação monstra um alerta que diz que houve sucesso no *login* e salta para a página dos eventos.

Caso o status seja diferente de 200 significa que aquela combinação de email e password não existe, logo não possibilita o *login* e mostra um alerta que diz que a combinação de email e password estão incorretos.

Depois da página do home, para além do login temos também a página de registo que, dentro da *tag head* usa também a tag style para definir o estilo da página. Usamos o mesmo estilo para o body, para os botões e para as caixas de texto.

Dentro da *tag body* temos todo o conteúdo dentro da *tag center* para poder termos tudo centrado no meio da página e dentro da *tag* h1 temos o título da página que diz "Registar".

Sendo que esta página consiste num formulário, usamos a *tag form* para criá-lo e assim, dentro desse *form* temos três *tags* h2, uma para escrever "Email", outra para escrever "Password" e outra para escrever "Nome" onde usamos três caixas de texto à frente desses textos para o utilizador introduzir o email, a password e o nome nos sítios respetivos.

Por baixo dessas caixas existe então um input do tipo botão que diz registar, ou seja, assim que o utilizador clicar nesse botão o programa vai chamar a nossa função registar, isto graças a propriedade onClick.

Dentro da *tag script* foi onde fizemos a nossa classe registar. No início da função existem dois *ifs*, um que que verifica se existe formação nas caixas de texto e caso não exista mostra um alerta. E o outro *if* verifica se o email está num formato válido.

A função contém um *fetch* do método *post* para fazer um *request* à API, para o body do *fetch* é enviado o valor da variável email, da variável password e da variável nome que correspondem à informação que está nas caixas de texto no momento em que o utilizador clica no botão login, o tipo de utilizador que é por defeito bombeiro e o ranking de confiança que vai por defeito a 10.

No fim da função, a partir do response existe um *if* que verifica se o status é igual a 302 e se sim mostra um alerta de erro, pois esse status significa que o email introduzido já existe na base de dados.

No fim existe ainda mais um *if* que verifica se o status é igual a 201, se sim, existe sucesso e a aplicação monstra um alerta que diz que houve sucesso no registo e salta de volta para a página home.

Depois da página do login, temos a página de eventos que, dentro da *tag head* usa a *tag style* para definir o estilo da página. Usamos o mesmo estilo para o body, para os botões e para as caixas de texto que nas páginas anteriores.

Dentro da *tag body* temos todo o conteúdo dentro da *tag center* para poder termos tudo centrado no meio da página e dentro da *tag* h1 temos o título da página que diz "Eventos não Classificados".

Esta página usa uma lista, ou seja, usa a *tag* e para cada elemento dessa lista usa o <li>, mas este <li> não é feito no html mas sim na função do javascript pois os números de eventos vai depender dos que ainda não estão classificados.

Por baixo da lista está um botão que diz sair e uma vez carregado leva o utilizador de volta à home page, visto este botão servir para o utilizador fazer logout.

Dentro do body existe a propriedade onload que irá fazer correr a nossa função evento e assim fazer com que a lista de eventos possa ser carregada para a página dos eventos assim que esta for acedida.

A função contém um *fetch* do método *get* para fazer um *request* à API. Dentro da *tag script*, onde fizemos a nossa classe eventos existe um *if* que verifica se o fetch recebeu um ok da API verificando se o status é igual a 200.

Depois do if que verifica se o status é igual a 200, graças ao get a função vai buscar todos os eventos não classificados esse código é corrido o numero de vezes necessários de acordo com o numero de evnetos não classificados e por cada evento que houver vai adicionar um à página com a tag <li> em que mostrará a data do evento e a localização dos cidadãos que enviaram os alarmes.

Esses eventos têm a propriedades de ao clicar num deles o site salta para a página que mostrará a informação referente a esse evento, ou seja, à nossa página Evento.html.

Depois da página dos eventos, temos a página de evento que é a página onde podemos ver a informação do evento em específico.

Dentro da tag head usamos a tag style para definir o estilo da página. Usamos o mesmo estilo para o body, para os botões e para as caixas de texto que nas páginas anteriores.

Dentro da *tag body* temos todo o conteúdo dentro da *tag center* para poder termos tudo centrado no meio da página e dentro da *tag* h1 temos o título da página que diz "Informação do Evento".

Esta página usa uma lista, ou seja, usa a tag <ul> e para cada informação do evento usa o <li>, mas este <li> não é feito no html mas sim na função do javascript pois a informação irá ser diferente de acordo com o evento selecionado.

Por baixo da lista estão dois botões, um deles diz incêndio e o outro diz não é um incêndio sendo que de acordo com o botão que for selecionado o evento será classificado como sendo um incêndio ou não.

Despois da classificação de acordo com o botão clicado aparecerá um alerta que terá uma mensagem a dizer que o evento foi classificado e este sairá da lista de eventos da página anterior.

Dentro do body existe a propriedade onload que irá fazer correr a nossa função start e assim fazer com que a informação do evento possa ser carregada para a página dos assim que esta for acedida.

A função contém um *fetch* do método *get* para fazer um *request* à API. Dentro da *tag script*, foi onde fizemos a nossa classe start, que no início vai verificar se o fetch recebeu um ok da API utilizando um *if* que confirma caso o status seja igual a 200.

Depois do *if* que verifica se o status é igual a 200, graças ao get a função vai buscar toda a informação do evento não classificado, e esse código é corrido o número de informações que esse evento possui e por cada informação é criado uma tag <li>.

As informações do evento, consiste na localização, na data do primeiro alarme, na data do último alarme, número de alarmes e no resultado probabilístico.

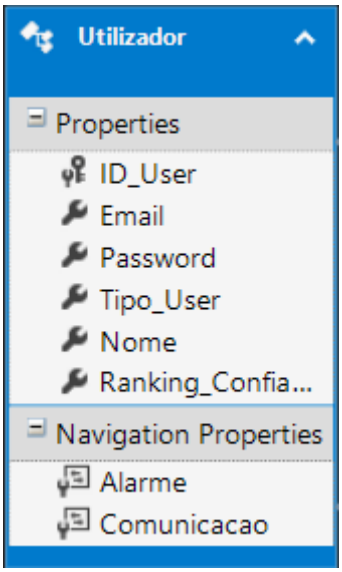
Back-end

O desenvolvimento do *back-end* passou pela criação de três recursos: uma **base de dados** para armazenamento da informação introduzida na aplicação (o nosso *front-end*) de modo a esta poder ser utilizada para criar respostas ás interrogações feitas a ela mesma; **dois servidores API *web*** capaz de comunicar quer entre si quer com a aplicação móvel quer com a base de dados, de modo a poder fazer pesquisas na mesma, consoante os pedidos enviados pela aplicação e tratar esses dados armazenados de modo a criar novo conteúdo relevante.

Back-end – Base de Dados

A **base de dados** foi desenvolvida em Microsoft SQL Server contendo 9 tabelas, das quais 5 tabelas estão a ser utilizadas correntemente para a implementação dos objetivos primários do projeto enquanto que 4 das tabelas foram criadas a pensar na implementação dos objetivos extra de classificação de imagens automaticamente e na implementação de comunicação direta entre o bombeiro no comando e um utilizador.

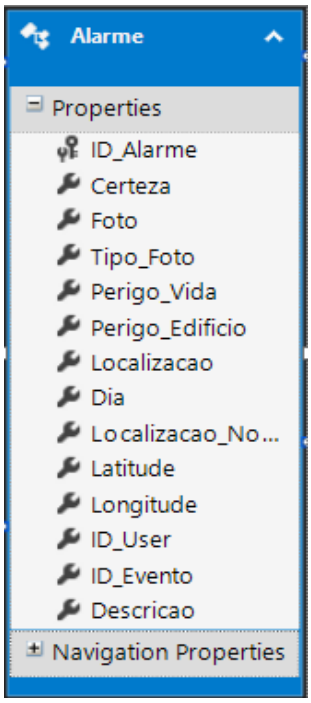
As 5 tabelas criadas com o intuito de implementação dos objetivos primários são:



1. **Tabela de "Utilizador":** onde se guarda a informação sobre o perfil dos utilizadores, criado quando estes se registam. Contém os campos de:
    a. **"Email",** do tipo varchar(150), que guarda o e-mail com que este se registou;
    b. **"Password",** do tipo varchar(150), que guarda a *password* para poder utilizar na autenticação do utilizador juntamente com o respetivo e-mail;
    c. **"Tipo_User",** do tipo varchar(50), que guarda a informação sobre o tipo de utilizador, ou seja, se é um cidadão ou um bombeiro;
    d. **"Nome",** do tipo varchar(50), que guarda o *username* do utilizador com que este se registou;
    e. **"Ranking_Confianca",** do tipo float, que guarda a informação sobre o ranking de confiança corrente do utilizador, sendo iniciado a 5 e sofrendo alterações consoante o número de alarmes que enviam e que acabam por ser classificados como sendo real ou não.

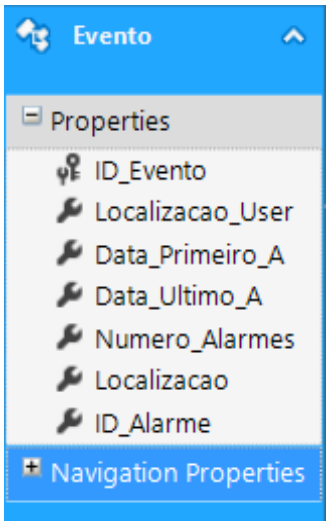
2. **Tabela de "Alarme":** onde se guarda a informação sobre os alarmes enviados pelos utilizadores. Contém os campos de:
    a. **"Certeza",** do tipo smallint, que guarda a certeza do utilizador em que caso é incêndio;

b. **“Foto”,** do tipo varbinary(max), onde se guarda a imagem que o utilizador enviou no alarme;
c. **“Tipo_Foto”,** do tipo bit, que guarda o resultado da classificação da imagem, sendo nulo se imagem não tiver sido enviada;
d. **“Perigo_Vida”,** do tipo smallint, que guarda o nível de perigo á vida que o utilizador achou;
e. **“Perigo_Edificio”,** do tipo smallint, que guarda o nível de perigo a edifícios que o utilizador achou;
f. **“Dia”,** do tipo datetime, que guarda a data de envio deste alarme;
g. **“Localizacao_Nome”,** do tipo, varchar(400), que guarda a localização do dispositivo móvel quando este envia alarme;
h. **“Latitude”,** do tipo float, que guarda a latitude em que o dispositivo móvel se encontrava quando enviou o alarme;
i. **“Longitude”,** do tipo float, que guarda a longitude em que o dispositivo móvel se encontrava quando enviou o alarme;
j. **“Descricao”,** do tipo varchar(500), onde guarda a informação recebida como descrição do que o utilizador está a ver.



k. Esta tabela faz ligação com duas outras: a tabela de Utilizador, na qual se encontra numa relação de **N-1**; e com a tabela de Evento, na qual se encontra numa relação de **N-0..1**.

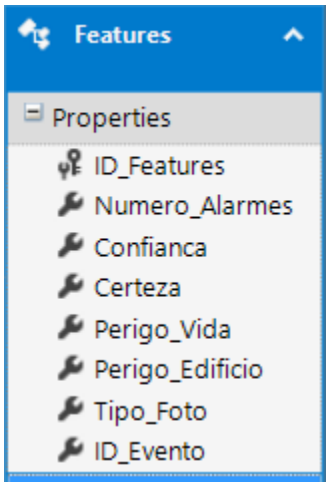
3. **Tabela de “Evento”:** onde se guarda a informação sobre os eventos, com base nas localizações dos alarmes recebidos. Contém os campos de:
   a. **“Localizacao_User”,** do tipo, varchar(400), que guarda a localização do dispositivo móvel quando este enviou alarme;
   b. **“Data_Primeiro_A”,** do tipo datetime, que guarda a data do primeiro alarme enviado que pertence a este evento;
   c. **“Data_Ultimo_A”,** do tipo datetime, que guarda a data do úlyimo alarme enviado que pertence a este evento;
   d. **“Numero_Alarmes”,** do tipo smallint, que guarda a informação do número total de alarmes que foram enviados sobre este evento.



   e. Esta tabela faz ligação com uma outra: a tabela de Alarme, na qual se encontra numa relação de **0..1-N**.

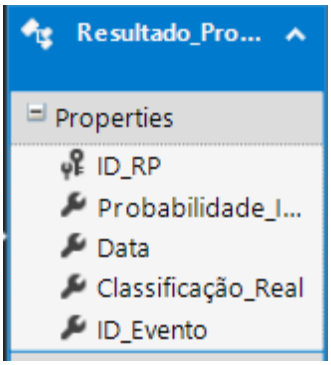
4. **Tabela de “Features”:** onde se guarda a informação sobre as variáveis necessárias para implementar a criação do modelo probabilístico de determinação de ser ou não um evento real e para previsão com base nesse modelo, acedendo á informação enviada no alarme e

servindo apenas para simplificar o processo de procura e acesso a esta informação. Contém os campos de:



a. **"Numero_Alarmes",** do tipo smallint, que é sempre igual a 1;
b. **"Confianca",** do tipo double, retirada do ranking de confiança do utilizador que enviou o alarme;
c. **"Certeza",** do tipo smallint, que guarda a certeza do utilizador em que caso é incêndio;
d. **"Perigo_Vida",** do tipo smallint, que guarda o nível de perigo á vida que o utilizador achou;
e. **"Perigo_Edificio",** do tipo smallint, que guarda o nível de perigo a edifícios que o utilizador achou;
f. **"Tipo_Foto",** do tipo bit, que guarda o resultado da classificação da imagem, sendo nulo se imagem não tiver sido enviada.
g. Esta tabela faz ligação com uma outra: a tabela de Evento, na qual se encontra numa relação de **N-1**.

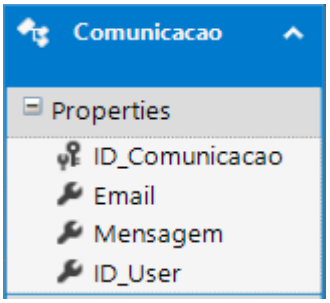
5. **Tabela de "Resultado_Probabilistico":** onde se guarda a informação sobre o resultado probabilístico da previsão do evento, em certa data. Contém os campos de:



   a. **"Probabilidade_Incendio",** do tipo float, onde se guarda a informação sobre a previsão probabilística de ser fogo;
   b. **"Data",** do tipo datetime, onde se guarda a data quando se executou previsão;
   c. **"Classificação_Real",** do tipo bit, onde se guarda informação sobre a classificação real do evento dada pelos bombeiros no comando.
   d. Esta tabela faz ligação com uma outra: a tabela de Evento, na qual se encontra numa relação de **N-0..1**.

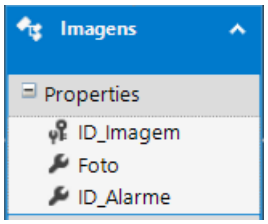
As 4 tabelas criadas com o intuito de implementação dos objetivos extra são:

6. **Tabela de "Comunicacao":** onde se guarda a informação sobre as mensagens trocadas diretamente entre o bombeiro e um cidadão, através de envio para da mensagem para o e-mail de destino. Contém os campos de:



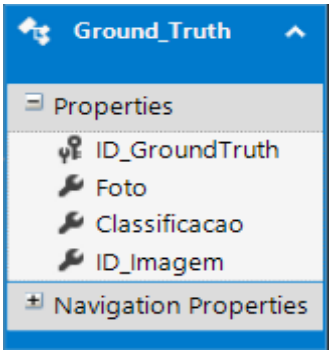
   a. **"Email",** do tipo varchar(150), onde se guarda o destino para qual será enviada a mensagem;
   b. **"Mensagem",** do tipo varchar(500), onde se guarda a mensagem a ser trocada.
   c. Esta tabela faz ligação com uma outra: a tabela de Utilizador, na qual se encontra numa relação de **N-1**.

7. **Tabela de "Imagens":** onde se guarda a própria imagem, retirada de cada alarme enviado pelo cidadão, com o intuito de centralizar e facilitar a procura de imagens para classificação. Contém o campo de:



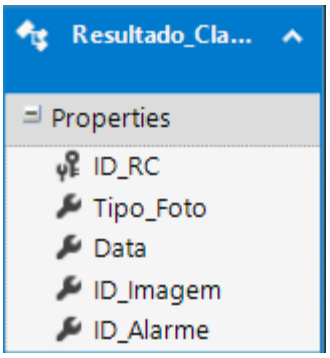
   a. **"Foto",** do tipo varbinary(max), onde se guarda a imagem que recolhe do alarme.
   b. Esta tabela faz ligação com uma outra: a tabela de Evento, na qual se encontra numa relação de **N-0..1**.

8. **Tabela de "Ground_Truth":** onde se guarda o *groundtruth* para a criação do modelo de classificação de imagens como sendo fogo ou fumo ou nenhum destes dois casos. Contém os campos de:



   a. **"Foto",** do tipo varbinary(max), onde se guarda cada imagem;
   b. **"Classificação",** do tipo bit, onde é guardada a informação real sobre se é ou não uma imagem de um fogo.
   c. Esta tabela faz ligação com uma outra: a tabela de Imagens, na qual se encontra numa relação de **N-0..1**.

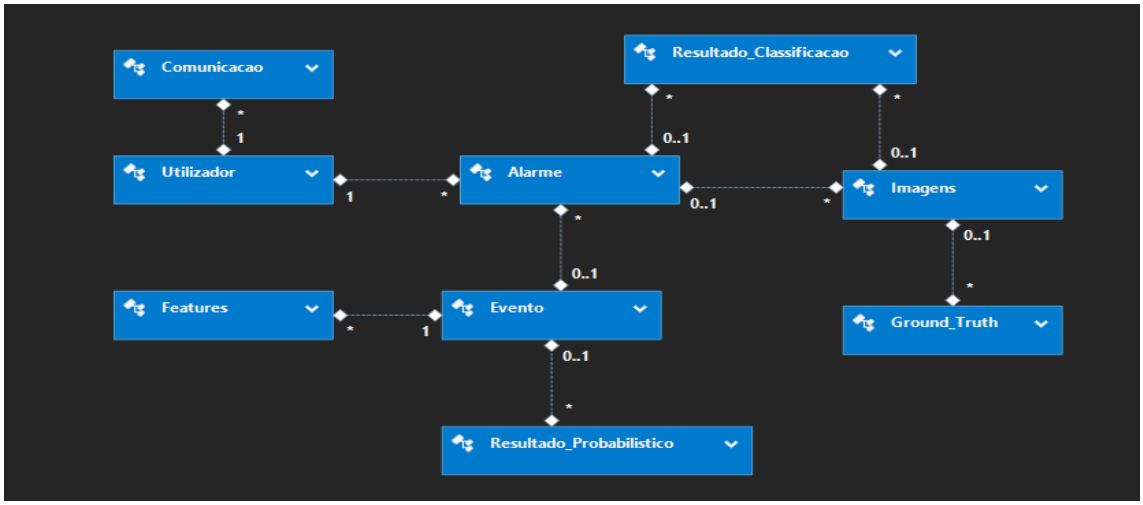
9. **Tabela de "Resultado_Classificacao":** onde se guarda a informação sobre a imagem e a sua respetiva classificação atribuída pelo modelo de classificação de imagens. Contém os campos de:



   a. **"Tipo_Foto",** do tipo bit, onde é guardada a informação sobre se é ou não uma imagem de um fogo;
   b. **"Data",** do tipo *datetime*, onde se guarda a data da classificação.
   c. Esta tabela faz ligação com duas outras: a tabela de Imagens, na qual se encontra numa relação de **N-0..1**; e com a tabela de Alarme, na qual se encontra numa relação de **N-0..1**.

Em comum entre todas as tabelas da base de dados é a existência de um campo de Identificação Única (ID), que é a respetiva chave primária da sua tabela, e daí ser um campo único, onde está implementado a propriedade de "*Identity Specification*" que permite a este campo ser automaticamente incrementado por 1 sempre que é introduzido uma nova instância na tabela.

Outro aspeto em comum é que as chaves secundárias que permitem fazer a ligação a outras tabelas são, em todas as tabelas criadas, os campos de identificação (ID) das tabelas a que estão ligadas.



Notar que nas imagens das tabelas se nota escrito alguns campos não utilizados. Estes campos seriam para guardar informação que não foi implementada no nosso projeto, mas sim para considerações futuras a ter sobre o desenvolvimento da aplicação móvel.

Um destes campos, por exemplo o campo **"Localização"** na tabela Alarme, foi introduzido para permitir ao cidadão identificar ele próprio onde o incêndio está a acontecer, caso este esteja relativamente fora do alcance da sua localização corrente, tomando este campo precedência sobre o que está correntemente a ser utilizado (**"Localização_Nome"**) para definição de que evento este alarme pertence. Logo, devido á não implementação desta consideração na aplicação móvel, este campo apenas aceita valores nulos e não é utilizado para nada. O mesmo se aplica para o campo **"Localização"** da tabela Evento.

Back-end – Servidores API *Web*

Ambos os **servidores API *web*** serão do tipo REST e será onde se irá desenvolver as regras de negócio e irá permitir interação com a base de dados, no nosso caso através da utilização de *Entity Framework*, o qual conecta e cria modelos da nossa base de dados, permitindo assim fazer a interação com as aplicações através da troca de *requests* e *responses*, em **HTTP**, executadas aos *urls* ocupados pelos servidores **[23]**.

O **servidor *web* REST API** foi desenvolvido no **Visual Studio 2017**, desenvolvido pela Microsoft, recorrendo ao uso das suas características para desenvolvimento de aplicações web, o **ASP.NET**, e as linguagens compatíveis com o mesmo, principalmente o **C#**. Neste criou-se, como falado anteriormente as regras de negócio para gerir as inquirições á base de dados e para manuseamento dos dados da mesma, criando novo conteúdo para apresentação a utilizadores.

Para permitir que este consiga fazer a manipulação de dados localizados na base de dados anteriormente criada, recorreu-se, como já foi dito, ao *Entity Framework*, um componente do *Visual Studio 2017*, com o intuito de ser um mapeador de objetos-relacionais, considerado como sendo umas das maiores ferramentas disponíveis para persistência dos mesmos.

Para tal é necessário criar uma conexão á base de dados criada anteriormente, através da configuração chamada de '`<connectionStrings>`'.

Estas regras encontram-se divididas em diferentes *urls*, sendo que devido a estarmos a desenvolver uma REST API, podem ter 4 métodos de *requests* aceitáveis, permitindo assim ao mesmo *url* servir para fazer 4 diferentes operações com respetivas regras de negócio diferentes. No caso do Visual Studio 2017, o *url* básico local é: **“ http://localhost:52344/api/ ”**, sendo inserido no final um identificador de qual o conjunto de operações a executar, determinando qual delas executar através do método enviado conjuntamente com este.

A estes *urls* irão ser efetuados *requests*, que se o método for ‘**POST**’, este deve incluir um *body*, ou seja, um corpo, no qual envia os valores de variáveis que serão necessárias. Nos **Controladores**, isto passa pela criação de um objeto dos respetivos modelos pertencente á base de dados a que o *Entity Framework* está conectado, com as variáveis equivalentes aos campos da respetiva tabela.

Por outro lado, se a estes *urls* forem efetuados *requests* com o método ‘**GET**’, este não inclui nenhum *body*, mas sim por transmitir os parâmetros a fornecer através do seu acresço ao respetivo *url*. Nos **Controladores**, isto passa pela leitura destes parâmetros e a sua utilização direta, como variáveis, para fazer as inquirições á base de dados.

Cada conjunto de operações implementadas e possíveis de executar pertencem a um **Controlador**, cujo nome é dado pelo identificador no *url* + *Controller*. Um exemplo é, no caso do conjunto de operações de Alarme, este Controlador passar a ser chamado de “AlarmeController”, expondo para o exterior (de modo a poder ser chamado pelas aplicações) o respetivo *url* de **“ http://localhost:52344/api/Alarme ”**.

Para o desenvolvimento das aplicações de *front-end*, quer a de dispositivo móvel quer a de apresentação ao bombeiro, foi necessário a criação de vários **Controladores**, e daí a disponibilização de vários *urls* expostos para o *front-end* poder chamar e enviar os pedidos necessários para o seu bom funcionamento. Dos **Controladores** aqui apresentados, 2 são únicos ao *front-end* extra de apresentação aos bombeiros, o **ProbabilidadeController** e o **ClassificarController**, responsáveis respetivamente pela apresentação dos resultados de previsão do modelo de classificação de eventos como sendo ou não fogo e permitir a classificação de evento como sendo real ou não. Estes são:

RegistarController:

- *URL* geral - **“ http://localhost:52344/api/Registar ”**.
- Apenas aceita um método no *request*: o método ‘**POST**’, logo apenas permite uma operação. Decidiu-se fazer o *request* através deste modo devido a este ser mais seguro do que fazer um ‘**GET**’, pois se fosse num deste método teríamos que escrever os dados no *url*, a que é fácil de aceder.
- Tem como objetivo introduzir o objeto criado (um Utilizador) na base de dados. Para tal recorre á utilização da *ConnectionString* para manipulação dos dados.
- Para um utilizador se poder registar, tem apenas uma única restrição: a de não se puder registar com um e-mail já presente na base de dados, ou seja, se o e-mail enviado no corpo for igual a um dos e-mails já presentes na base de dados.
- Se esta restrição não se aplicar ao objeto de Utilizador recebido, então guardará esse objeto na base de dados.
- Este método ‘**POST**’, envia como resposta uma mensagem (*response*) do tipo `HttpResponseMessage`, podendo esta ter 3 tipos: uma de sucesso para quando conseguiu inserir na base de dados o utilizador, de *status code* 201 (*Created*) mais o objeto criado;

duas de insucesso: uma para quando se dá a restrição, gerando um *status code* 302 (*Found*) com uma mensagem a informar que encontrou o e-mail inserido na base de dados e outra mensagem para quando as variáveis introduzidas no *body* não correspondem ás esperadas, que gera um *status code* 400 (*BadRequest*).

LogInController:

- *URL* geral - **“ http://localhost:52344/api/LogIn ”**.
- Apenas aceita um método no *request*: o método ‘**POST**’, logo apenas permite uma operação. Decidiu-se fazer o *request* através deste modo devido a este ser mais seguro do que fazer um ‘**GET**’, pois se fosse num deste método teríamos que escrever os dados no *url*, a que é fácil de aceder.
- Tem como objetivo verificar que o objeto criado (parte de um Utilizador, neste caso apenas o e-mail e a *password*) tem equivalente na base de dados. Para tal recorre á utilização da *ConnectionString* para manipulação dos dados.
- Para um utilizador poder dar *login*, tem apenas uma única restrição: a de o e-mail e a *password* que enviou no *body* serem iguais aos de uma instância de utilizador na tabela onde estes se encontram guardados.
- Se esta restrição se aplicar ao objeto parcial de Utilizador recebido, então responderá com sucesso de *login*.
- Este método ‘**POST**’, envia como resposta uma mensagem (*response*) do tipo `HttpResponseMessage`, podendo esta ter 3 tipos: uma de sucesso para quando conseguiu encontrar um utilizador com os mesmos dados que recebeu, de *status code* 200 (*OK*) e a instância da base de dados encontrada; duas de insucesso: uma para quando não encontra nenhuma instância na base de dados com os parâmetros recebidos, gerando um *status code* 404 (*NotFound*) com uma mensagem a informar que não encontrou o e-mail enviado e outra mensagem para quando as variáveis introduzidas no *body* não correspondem ás esperadas, que gera um *status code* 400 (*BadRequest*).

AlarmeController:

- *URL* geral - **“ http://localhost:52344/api/Alarme ”**.
- Aceita dois métodos no *request*: um método ‘**GET**’ e um método ‘**POST**’, ambos partilhando o mesmo *url*, mas o método ‘**GET**’ receber como parâmetro no *url* o Id de Utilizador.
- Exemplo *URL* final de método ‘**GET**’ – **“ http://192.168.1.67:3000/api/Alarme/2 “**, onde ‘2’ é o Id do Utilizador.
- **No caso de operações através do método** ‘**GET**’ com parâmetro recebido em *url*, este tem como objetivo a apresentação de uma lista dos últimos três alarmes enviados pelo utilizador identificado através do Id recebido por *url*. Para tal recorre á utilização da *ConnectionString* para manipulação dos dados.
- Para poder retornar essa lista, tem apenas de fazer uma única pesquisa á base de dados: encontrar todas as instâncias da respetiva tabela (“**Alarme**”) que tenham um Id de Utilizador igual ao recebido como parâmetro no *url* e retornar os últimos 3.
- Este método ‘**GET**’, envia então como resposta uma mensagem (*response*) do tipo `IEnumerable<Alarme>`, em que `Alarme` é o modelo da tabela presente na base de dados, e podendo retornar 2 tipos de lista: uma lista três objetos, ou seja, das últimas três

instâncias encontradas com Id de Utilizador igual á do recebido pelo *url* (caso de sucesso); e uma outra lista que não contém qualquer objeto, logo uma lista vazia, no caso de não encontrar na base de dados nenhuma instância com esse Id (caso de insucesso).

- **No caso de operações através do método** '**POST**', que recebe os parâmetros das variáveis a utilizar através do *body*, pertencendo estas á tabela de "**Alarme**". Para tal recorre á utilização da *ConnectionString* para manipulação dos dados.
- Tem como objetivo a inserção na base de dados dos alarmes enviados por cada utilizador e na sua divisão em eventos, quer através da criação de um novo evento quer através da modificação de um evento anterior (soma do número total de alarmes e modificação da data do último alarme enviado por utilizador), transferindo ainda a informação necessária à criação de um modelo de determinação probabilístico através da transferência dos parâmetros necessários para uma tabela á parte (a tabela "**Features**"). Finalmente, deve recorrer ao *url* disponibilizado pelo outro servidor *web* REST API para efetuar uma previsão de probabilidade de ser um evento real ou não, através do modelo de determinação probabilístico, guardando os resultados na respetiva tabela. Esta operação permite ainda guardar uma fotografia recebida como parâmetro na tabela de "**Imagens**".
- Para atingir estes objetivos, esta recorre a guardar a informação sobre o alarme enviado nos parâmetros do *body*, na base de dados, utilizando de seguida esses parâmetros para fazer uma pequena pesquisa para determinar o Id que foi atribuído à instância na base de dados. Isto deve-se a, como referido anteriormente, o Id ser atribuído automaticamente pelo SQL Server.
- Verifica se os parâmetros que recebeu incluem uma foto, ao qual se estes tiverem cria uma nova instância da tabela "**Imagens**" ao qual atribui o Id do alarme enviado e a própria fotografia.
- De seguida, vai classificar a que evento o alarme enviado pertence, através da localização do dispositivo móvel enviada no *body* e da data em que este foi enviada. Para tal faz um procura na tabela de "**Evento**" por todos os eventos localizados na mesma localização que a enviada nos parâmetros.
- Se não tiver encontrado nenhum evento então cria um novo com a mesma data que a do alarme enviado, a sua localização, o número de alarmes (igual a 1) e, como foi apenas um alarme, sem data de último alarme.
- Se tiver encontrado algum evento, vai depois a cada um dos eventos que encontrou e verifica se a data do primeiro alarme é nula, para o qual cria novo evento, ou se é diferente de nula. Se for diferente de nula, irá por fim verificar se a data do dia em que enviou alarme e a data do primeiro alarme têm ou não uma diferença menor que '15'. Tem que ser menor que '15' devido a este ser o espaço de tempo definido para ser considerado com um único evento.
- No caso da diferença entre a data do dia de envio e a data do primeiro alarme for menor que 15, então pertencem ao mesmo evento, necessitando apenas de modificar os valores guardados por essa instância, somando 1 ao número de alarmes e modificando a data do último alarme para a data de envio de alarme nos parâmetros.
- Caso contrário, deve-se prosseguir á criação de um novo evento.
- Após a criação ou modificação do evento, deve-se atribuir na tabela de "**Alarme**" qual o Id de Evento a que este alarme pertence.
- Após o tratamento da atribuição de evento, este deve passar os parâmetros necessários à previsão do modelo probabilístico para a tabela dedicada, a "**Features**", para qual necessita do nível de confiança do Utilizador que enviou o alarme. Para tal vai procurar

na tabela de "**Utilizador**" através do Id de Utilizador que recebeu nos parâmetros, pois para poder enviar alarme um utilizador tem que estar autenticado, e selecionar o nível de confiança corrente do Utilizador.

- Após esta pesquisa, cria uma instância da tabela "**Features**" com os respetivos parâmetros necessários a serem preenchidos.
- No final irá criar um cliente HTTP para consumir serviços de um outro servidor *web*, também ele um REST API, para o qual o cliente HTTP irá chamar o URL – "`http://localhost:55555/api/predict/{id}`", em que '{id}' é o Id do Evento a executar a previsão sobre.
- Na resposta enviada a este cliente HTTP vem o valor da probabilidade de ser ou não um evento real. Esta resposta é depois passada como parâmetro a introduzir na tabela juntamente com o ID do Evento de que foi executado a previsão e a data da altura em que foi executada.
- Este método '**POST**', envia como resposta uma mensagem (*response*) do tipo `HttpResponseMessage`, podendo esta ter 3 tipos: uma de sucesso para quando conseguiu introduzir todos os dados nas respetivas tabelas, de *status code* 201 (*Created*); duas de insucesso: uma para quando não conseguiu introduzir todos os dados nas tabelas, gerando um *status code* 400 (*BadRequest*) com uma mensagem a informar que não encontrou um evento correspondente e outra mensagem para quando as variáveis introduzidas no *body* não correspondem ás esperadas, que gera um *status code* 400 (*BadRequest*).

EventosController:

- *URL* geral - **" http://localhost:52344/api/Eventos "**.
- Aceita dois métodos no *request*: um método '**GET**' e um método '**POST**', ambos partilhando o mesmo *url* devido a nenhum deles levar necessitar de um parâmetro específico necessário ser introduzido fora do *body*.
- **No caso de operações através do método** '**GET**', tem como objetivo a apresentação dos últimos cinco eventos criados, retornando para tal os primeiros cinco elementos de uma lista de objetos ordenados através do número de alarmes que cada um engloba, com o intuito de apresentá-los aos utilizadores (aos Cidadãos). Para tal recorre á utilização da *ConnectionString* para manipulação dos dados.
- Para poder retornar esta lista de elementos, tem apenas de fazer uma única pesquisa á base de dados: encontrar todos as instâncias da tabela de "**Evento**" que tenham uma diferença de dias desde que foram criados (data do primeiro alarme deste evento) e o dia de hoje seja menor que 30 (valor de diferença é de 30 dias, devido a considerar que todos os alarmes enviados dentro de 15 dias é o mesmo evento e deixar uma folga de relevância para o caso de não aparecer mais nenhum evento), ordenando-os por ordem descendente devido a queremos os elementos com valor de números de alarmes mais elevados.
- Este método '**GET**', envia então como resposta uma mensagem (*response*) do tipo `IEnumerable<Evento>`, em que `Evento` é o modelo da tabela presente na base de dados, e podendo retornar 2 tipos de lista: uma lista de até cinco objetos, quando encontra instâncias que datas encaixem dentro do espaço de tempo definido (30 dias), no caso de sucesso; e uma outra lista que não contém qualquer objeto, logo uma lista vazia, no caso de não encontrar na base de dados nenhuma instância dentro desse espaço de tempo, no caso de insucesso.

- **No caso de operações através do método** '**POST**', tem como objetivo retornar uma indicação, através do *status code* enviado na resposta, se encontrou ou não um evento na localização enviada no *body*, correspondente á localização corrente do dispositivo móvel. Para tal recorre á utilização da *ConnectionString* para manipulação dos dados.
- Para poder decidir se existe um evento na sua localização ou não, tem apenas de fazer uma única pesquisa á base de dados: percorrer todas as instâncias na base de dados da tabela de "**Evento**" até encontrar uma que seja igual á localização que recebe como parâmetro a partir do *body* e que esteja num espaço de tempo inferior a 15 dias em comparação com a data do primeiro alarme enviado que pertença a este evento 30 (valor de diferença é de 15 dias, devido a considerar que todos os alarmes enviados dentro de 15 dias pertencem ao mesmo evento para a mesma localização).
- Este método '**POST**', envia como resposta uma mensagem (*response*) do tipo `HttpResponseMessage`, podendo esta ter 3 tipos: uma de sucesso para quando conseguiu encontrar um utilizador com os mesmos dados que recebeu, de *status code* 200 (*OK*) e uma lista de instâncias da base de dados encontrada; duas de insucesso: uma para quando não encontra nenhuma instância na base de dados igual á do parâmetro recebido, gerando um *status code* 404 (*NotFound*) com uma mensagem a informar que não encontrou nenhum evento e outra mensagem para quando as variáveis introduzidas no *body* não correspondem ás esperadas, que gera um *status code* 400 (*BadRequest*).

FeaturesController:

- *URL* geral - **"** **http://localhost:52344/api/Features** **"**.
- Aceita dois métodos no *request*: um método '**GET**' e um método '**GET**' com envio de parâmetro através do *url*, ambos sem qualquer *body*.
- Exemplo *URL* final de método '**GET**' com parâmetro transmitido através de *url* – **"** **http://192.168.1.67:3000/api/Features/33** **"**, onde '33' é o Id do Evento a apresentar.
- **No caso de operações através do método** '**GET**' **com parâmetro recebido em *url***, este tem como objetivo a apresentação de uma lista das *features*, ou seja, das variáveis necessárias para a criação de um modelo de probabilidade determinístico de ser um evento real ou não, enviadas pelos utilizadores, do evento de qual recebe o identificador (Id), presente no *url*. Para tal recorre á utilização da *ConnectionString* para manipulação dos dados.
- Para poder retornar essa lista, tem apenas de fazer uma única pesquisa á base de dados: encontrar todas as instâncias da respetiva tabela ("**Features**") que tenham um Id de Evento igual ao recebido como parâmetro no *url*.
- Este método **GET**' com parâmetro recebido em *url*, envia então como resposta uma mensagem (*response*) do tipo `IEnumerable<Features>`, em que `Features` é o modelo da tabela presente na base de dados, e podendo retornar 2 tipos de lista: uma lista de todos os objetos, ou seja, de todas as instâncias encontradas com Ids de eventos iguais ás do recebido (caso de sucesso); e uma outra lista que não contém qualquer objeto, logo uma lista vazia, no caso de não encontrar na base de dados nenhuma instância com esse Id (caso de insucesso).
- **No caso de operações através do método** '**GET**' **sem parâmetro recebido em *url***, este tem como objetivo a apresentação de uma lista dos eventos que ainda não foram classificados como sendo ou não um evento real, por parte dos e para apresentar aos

bombeiros, desde o mais recente para o mais antigo evento. Para tal recorre á utilização da *ConnectionString* para manipulação dos dados.

- Para poder retornar essa lista, tem de fazer duas pesquisas á base de dados: uma para encontrar todas as instâncias da respetiva tabela ("**Resultados_Probabilisticos**") cujo valor do campo "Classificacao_Real", onde a base de dados armazena a informação sobre a classificação real de cada evento, seja nulo e retornar os valores distintos dos respetivos Ids de Evento; outra pesquisa para encontrar todas as instâncias da respetiva tabela ("**Evento**") para cada um dos Ids que pertençam ao grupo de Ids retornados pela pesquisa anterior e retornar o respetivo objeto (instância), ordenando a lista por ordem descendente tendo em conta o Id de Evento, ou seja, do evento mais recente para o mais antigo.
- Este método '**GET**' sem parâmetro recebido em *url*, envia então como resposta uma mensagem (*response*) do tipo `IEnumerable<Evento>`, em que `Evento` é o modelo da tabela presente na base de dados, e podendo retornar 2 tipos de lista: uma lista de todos os objetos, ou seja, de todas as instâncias encontradas com Ids de eventos iguais aos Ids de evento que ainda não foram classificados, ordenados decrescentemente por ordem de criação (caso de sucesso); e uma outra lista que não contém qualquer objeto, logo uma lista vazia, no caso de não encontrar na base de dados nenhuma instância com esses Ids (caso de insucesso).

ProbabilidadeController:

- *URL* geral - **"** **http://localhost:52344/api/Probabilidade** **"**.
- Apenas aceita um método no *request*: o método '**GET**', logo apenas permite uma operação. Isto obriga a que seja acrescido ao *url* o parâmetro a transmitir á API para este fazer as interrogações á base de dados, neste caso o Id do Evento em questão.
- Exemplo *URL* final – **"** **http://192.168.1.67:3000/api/Probabilidade/33** **"**, onde '33' é o Id do Evento a apresentar.
- Tem como objetivo a apresentação dos resultados de previsão do modelo de classificação de eventos como sendo ou não fogo, retornando para tal apenas o primeiro elemento de uma lista de objetos, com o intuito de retornar o valor da probabilidade de certo evento ser ou não um incêndio real. Para tal recorre á utilização da *ConnectionString* para manipulação dos dados.
- Para poder retornar esse elemento, tem apenas de fazer uma única pesquisa á base de dados: encontrar todas as instâncias da tabela que tenham um Id de Evento igual ao recebido como parâmetro no *url*, ordenando-os por ordem descendente devido a queremos apenas o valor da última previsão efetuada, tendo em conta as datas em que estas foram feitas.
- Este método '**GET**', envia então como resposta uma mensagem (*response*) do tipo `IEnumerable<Resultado_Probabilistico>`, em que `Resultado_Probabilistico` é o modelo da tabela presente na base de dados, e podendo retornar 2 tipos de lista: uma lista de um único objeto, quando encontra instâncias com Ids de eventos iguais ás do recebido (caso de sucesso); e uma outra lista que não contém qualquer objeto, logo uma lista vazia, no caso de não encontrar na base de dados nenhuma instância com esse Id (caso de insucesso).

ClassificarController:

- *URL* geral - **“ http://localhost:52344/api/Classificar ”**.
- Apenas aceita um método no *request*: o método ‘**POST**’, logo apenas permite uma operação. Devido á necessidade de saber qual o Evento a classificar, levou-se a adotar a opção de mandar o Id do Evento correspondente através do *url*, enviando através do *body* o parâmetro a modificar.
- Exemplo *URL* final – **“ http://localhost:52344/api/Classificar/33 “**, onde ‘33’ é o ID do Evento a classificar.
- Tem como objetivo a classificação de um evento como sendo real ou não, alterando para tal os objetos (instâncias da tabela de “**Resultados_Probabilisticos**”) na base de dados. Para tal recorre á utilização da *ConnectionString* para manipulação dos dados.
- Para poder classificar um evento, tem apenas de fazer uma única pesquisa á base de dados: encontrar todos as instâncias desta tabela que tenham um Id de Evento igual ao recebido como parâmetro no *url* e para cada instância encontrada que corresponda a esta procura deve modificar o campo de “Classificacao_Real” pelo valor recebido no *body*, guardando as alterações feitas á base de dados.
- Posteriormente a esta modificação, deve-se mudar o nível de confiança em cada utilizador que enviou alarme para este evento, diferenciando o comportamento dependentemente de este ter sido classificado como evento real ou não. Se foi considerado real, então somará ao nível de confiança corrente um valo de 0,1. Se não foi considerado real, então subtrairá ao nível de confiança corrente um valor de 0,1.
- Para tal modificação, irá efetuar uma pesquisa á tabela “**Alarme**” da base de dados que lhe permita retornar os Ids distintos dos Utilizadores que enviaram os alarmes que correspondem ao Id de Evento recebido como parâmetro no *url*. Esse nível de confiança fica limitado a valores inferiores ou iguais a 10 e superiores ou igual 0.
- Este método ‘**POST**’, envia como resposta uma mensagem (*response*) do tipo `HttpResponseMessage`, podendo esta ter 2 tipos: uma de sucesso para quando conseguiu identificar na base de dados uma instância de uma previsão para o Id do Evento recebido, de *status code* 200 (*OK*); outra de insucesso para quando não conseguiu identificar na base de dados uma instância de uma previsão para o Id do Evento recebido, gerando um *status code* 404 (*NotFound*) com uma mensagem a informar que não encontrou previsões para esse Id inserido na base de dados.

Back-end – Servidor API FLASK

O **servidor *web* FLASK REST API** foi desenvolvido no **Visual Studio 2017**, desenvolvido pela Microsoft, recorrendo ao uso das suas características para desenvolvimento de aplicações web e de um dos seus componentes, o **Flask**, e a linguagem compatível com o mesmo, o **Python**, que permite utilizar esta linguagem de programação para a criação de um servidor *web* e de uma respetiva REST API. Neste criaram-se as regras de negócio para fazer o treino do modelo de determinação probabilística de ser um evento real ou não e da respetiva previsão probabilística.

Para tal recorreu-se ao uso dos seguintes pacotes da linguagem de programação, **Python**: o `flask`, um *micro web framework* simples e fácil de implementar, sem bibliotecas; `sklearn` (scikit-learn), uma biblioteca que permite implementar *machine learning*, que inclui vários algoritmos pré-definidos; `numpy`, biblioteca para computações cientificas; `pandas`, bibliotecas para manipulação de dados e análises dos mesmos; `pyodbc` (Python SQL Driver), que permite fazer inquirições à base de dados através de comandos SQL; `math`, biblioteca que providencia funções para cálculos matemáticos; `csv`, biblioteca para importe e exporte de documentos no formato de valores separados por vírgulas.

Devido a não possuirmos um ***Dataset*** real para treinarmos o modelo, recorremos ao processo de criarmos nós próprios um ficheiro *.csv* com que iremos trabalhar, de 9998 linhas. Este ficheiro segue regras simples para determinação da classificação real do evento. Essas são:

- Criação de 13 colunas, seis das quais são as colunas pertencente aos parâmetros a que queremos aplicar o modelo, 6 que são o resultado de o dado dessa coluna completar ou não a regra de seleção (se sim, coluna recebe dado 1, se não recebe dado 0) e uma coluna final, que é a classificação real atribuída.
- Para as primeiras 6 colunas existem uma restrição. Os valores de Número de Alarmes estão entre 1-50, enquanto que todas as outras colunas estão entre 1-10. Estes valores são gerados automaticamente e aleatoriamente.
- Para decidir o valor das próximas 6 colunas, é dependente de terem ou não sucesso na restrição associada. Essas são: se o número total de alarmes for superior a 10, então é real; se a média do nível de certeza for superior a 4, então é real; se a média do nível de perigo de vida for superior a 4, então é real; se a média do nível de perigo a edifícios for superior a 6, então é real; se o tipo de foto for 1 (*true*), então é real; se a média do nível de confiança for superior a 5, então é real.
- Para decidir o valor da coluna de classificação real soma os dados das 6 colunas anteriores e se essa soma for superior a 3, então classifica evento como ser real, caso contrário classifica como sendo evento falso.

Nesta REST API, para definir um conjunto de regras (operações) é necessário definir primeiro o caminho, a *Route*, em que este vai poder ser chamado, através do *url*. Também nesta será necessário indicar quais os métodos que esta aceita, dos 4 possíveis.

O URL geral será: " **http://localhost:52344** ".

Como falado anteriormente, implementou-se apenas 2 operações:

Operação para fazer treino do modelo para determinação da probabilidade de ser ou não evento real:

- Definição de qual o *url* a expor para consumo por parte dos clientes, através da definição de `@app.route()` - " **http://localhost:52344/api/train** " através do método '**GET**'.
- O modelo probabilístico escolhido foi o de ***Linear Regression***. No nosso projeto, este modelo vai ter uma base de aprendizagem estática, o que foi construído no ficheiro CSV, e uma parte de aprendizagem dinâmica, com a qual ele vai melhorar a aprendizagem ao longo do tempo. Este melhoramento do modelo ao longo do tempo deve-se á utilização dos eventos classificados, por parte dos bombeiros, que são, depois de efetuados o seu tratamento dos dados, juntos á informação presente no ficheiro CSV e utilizados, então, para executar o treino (aumentando o número de instâncias a serem usadas para a criação do modelo, melhora o *score*, consoante o maior número de instâncias).
- Para construir o modelo tem de se seguir os seguintes passos gerais: juntar informação em único *dataset*; fazer a divisão desse *dataset* em *dataset* de treino e de teste; aplicar o modelo de *Linear Regression* da biblioteca '**sklearn**', ajustado às colunas e respetiva classificação real; persisti-lo para fora desta operação de modo a poder ser chamado noutro método.
- Para juntar a informação em um único *dataset*, é necessário fazer o tratamento da mesma. Para tal faz-se:

- Nesta operação começamos por importar o ficheiro CSV, chamado `'teste.csv'`, que contém os dados principais já tratados e prontos a serem usados para executar o treino do modelo, tendo apenas de retirar as colunas que não contêm informações necessárias - as de *true/false* pelas colunas geradas automaticamente, terem ou não, cumprido certas restrições. Os dados deste ficheiro são então atribuídos a um *dataset*.
- De seguida, inicia-se o Python SQL Driver, de modo a poder-se fazer inquirições á base de dados.
- Terá de se ir buscar á base de dados todos os valores das variáveis necessárias, por evento, aos quais se irá descobrir as médias por coluna, exceto à coluna de número de alarmes (devido a ser retirada da tabela de "**Evento**", em vez da tabela de "**Features**"), para obter as variáveis necessárias para executar o treino do modelo de *Linear Regression,* juntando esta informação relativa às médias como sendo uma única linha num *dataset*, que irá conter toda esta informação para cada evento.
- Os valores retirados durante a pesquisa á base de dados por evento pode, em certos casos, gerar colunas que não contenham valores, mais especificamente, a coluna de "**Tipo_Foto**". Neste caso, é lhe atribuído o valor de '0', passando a ser tratada como se tivesse sido atribuído um valor de classificação de foto igual a falso.
- É também necessário que os valores que lhe são retirados possam ser utilizados diretamente pelo modelo *Linear Regression*, sendo assim necessário fazer a conversão dos estados do tipo *boolean* de *true/false* para valores inteiros do tipo *boolean* de 0/1.
- É ainda necessário fazer uma escolha dos eventos a utilizar dos retirados da base de dados, sendo que são apenas necessários os que contenham valores diferentes de nulo, na coluna/campo de "**Classificacao_Real**", devido a serem estes que foram classificados pelos bombeiros.
- Tal como se notou na coluna de "**Tipo_Foto**" é também necessário que os valores presentes na coluna de "**Classificacao_Real**" possam ser utilizados diretamente pelo modelo, sendo por isso necessário, também nesta coluna, fazer a conversão de estados do tipo *boolean* de *true/false* para valores inteiros do tipo *boolean* de 0/1.
- Apenas após de fazer estes passos se poderá juntar o *dataset* do ficheiro CSV importado com o *dataset* criado com base nas pesquisas á base de dados que engloba as médias das variáveis (*features*), com o número de alarmes enviados, por evento.

- Depois de se obter um único *dataset*, prossegue-se então com a sua divisão em *datsets* para treino e testes. Estes são divididos em 80% dos dados (linhas) pertencem ao *dataset* treino e 20% pertencem ao *dataset* teste. Por sua vez, estes *datasets* são divididos por colunas: um *dataset* de *values* que contem as colunas com interesse para obter a classificação de cada objeto e um *dataset* de *labels* que contem apenas a coluna que contem os valores da classificação real para os objetos do *dataset* de *values*.
- Deve-se então aplicar o modelo de *Linear Regression* (LN) importado da biblioteca '*sklearn'*, aos quais se deve ajustar, ou seja, fazer 'LN.fit()', do *dataset* de *values* e do *dataset* de *labels*. Tal como em: '`lm.fit(train_values,train_labels)`'.
- Finalmente, tem que se persistir o método para fora desta operação de modo a poder ser chamado noutro método. Ou seja, deve-se exportar este método como um ficheiro para

que este possa, por sua vez, ser chamado noutra operação. Este ficheiro vai ser exportado em '.pkl'. Para tal é necessário utilizar o método '`joblib`', proveniente da biblioteca '*sklearn*'. Tal como em: '`joblib.dump(LM, 'Train_Linear_Regression_Model.pkl')`'.

- Esta método '**GET**' retorna como resposta uma mensagem (*response*), podendo retornar 2 tipos de mensagens, ambas em JSON: uma de sucesso, que retorna a mensagem de "`Treino de dados efetuado com sucesso. Para confirmação ver data ficheiro criado.`" e uma de insucesso que retorna a mensagem de "`Dados incorretos.`". Para enviar estas mensagens através de JSON recorre-se ao seguinte método: `jsonify()`.

Operação para fazer previsão com base no modelo de probabilidade:

- Definição de qual o *url* a expor para consumo por parte dos clientes, através da definição de `@app.route()` - " **http://localhost:52344/api/predict/<id>** " através do método '**GET**', onde '<id>' é o Id do Evento a prever.
- Este *url* é consumido pela REST API de servidor *web* em C#, cada vez que um novo alarme é enviado, permitindo obter vários valores de previsões para o mesmo Id de evento enviado, consoante o número de alarmes enviados, divididos pela sua data.
- Esta operação passa pela utilização do modelo de *Linear Regression* treinado e persistido na operação anterior e, através do Id de Evento recebido no *url*, tornar os parâmetros necessários para à previsão na composição correta para executá-la.
- Para executar esta previsão, é necessário selecionar os parâmetros necessários para tal de um evento determinado, iguais aos parâmetros treinados, as chamadas *features*, anteriormente definidas e passar estas para um *dataset* único. Neste *dataset* as *features* deverão então ser tratadas de modo a poderem ser utilizadas. Deve-se então importar o ficheiro que contem o modelo persistido e executar a previsão com as *features* obtidas enviando o resultado como resposta.
- Para a criação do *dataset* que conterá com as *features* finais, já tratadas e prontas a serem utilizadas para a operação de previsão é necessário:
    - Permitir que linguagem Python possa fazer pesquisas diretas à base de dados utilizada, através do Python SQL Driver.
    - Selecionar as *features* necessárias da base de dados através das pesquisas por Id de Evento, igual ao recebido como parâmetro no *url*.
    - Executar uma área de exceção para o caso do Id de Evento recebido não exista na base de dados, logo não se tem nem alarmes nem *features* dele, enviando imediatamente uma resposta (*response*) que retorna uma mensagem, em JSON, que indica um caso de insucesso com a mensagem de `"Id inválido. Escolha outro."`. Caso contrário, criar um *dataset* de dados não tratados.
    - Tal e qual como na operação anterior, os valores retirados durante a pesquisa á base de dados por evento pode, em certos casos, gerar colunas que não contenham valores, mais especificamente, a coluna de "**Tipo_Foto**". Neste caso, é lhe atribuído o valor de '0', passando a ser tratada como se tivesse sido atribuído um valor de classificação de foto igual a falso.
    - É também necessário que os valores que lhe são retirados possam ser utilizados diretamente pelo modelo *Linear Regression*, sendo assim necessário fazer a conversão dos estados do tipo *boolean* de *true/false* para valores inteiros do tipo *boolean* de 0/1.
    - Irá-se então irá descobrir as médias por coluna dos resultados da pesquisa, exceto à coluna de número de alarmes (devido a ser retirada da tabela de "**Evento**", em

vez da tabela de "**Features**"), para obter as variáveis necessárias para executar a previsão com o modelo de *Linear Regression*, juntando esta informação das médias como sendo uma única instância dum *dataset*.

- Finalmente deve-se importar o ficheiro que contém a informação sobre o modelo, guardado em '.pkl', previamente criado/alterado sempre que é corrida a operação de executar treino do modelo. Para tal é necessário recorrer á utilização do método de '`joblib`', proveniente da biblioteca '*sklearn*'. Exemplificado em: '`lm = joblib.load('Train_Linear_Regression_Model.pkl')`'.
- Para além da resposta enviada quando o Id de Evento recebido não corresponde a nenhum dos Eventos na base de dados, uma mensagem de insucesso, este pode ainda lançar outras duas respostas (*responses*), ambas em JSON, que no caso de sucesso retorna como resposta uma mensagem que contém uma lista de um único elemento contendo o valor da previsão do obtida através do modelo de *Linear Regression*, e uma no caso de insucesso que retorna como resposta uma mensagem `"Dados incorretos."`.

Testes/Resultados

Uma lista dos testes automáticos executados ao *back-end* e ao *front-end*, através de software próprio, no nosso caso o Postman e o Expo, até ao sucesso dos *requests*, de acordo com a lógica desenvolvida.

Esses testes automáticos recorrem ao uso do software próprio para controlar e executar os testes, fazendo depois a comparação dos resultados obtidos com os resultados reais esperados.

Nestes testes existem duas abordagens: teste á interface gráfica do utilizador e teste baseado no código **[24]**.

- **Teste á interface gráfica do utilizador:** ferramentas que simulam funcionalidades e ações de um utilizador, aplicável a qualquer aplicação com interface gráfica, e com a vantagem de exigir menos codificação, porém mais difícil fazer a manutenção (como, por exemplo, quando a interface é alterada).

- **Teste baseado no código:** executa-se testes às unidades de código desenvolvido para determinar se todas as secções do código estão a funcionar da forma esperada, para diversas circunstâncias.

Testes

Testes aos *front-end:*

Uma lista dos testes executados ao *front -end*, através do Expo, de todos os *requests* errados, de acordo com a lógica desenvolvida, para retornar todos os alertas que visam avisar um utilizador que os parâmetros introduzidos estão errados e que é necessário corrigi-los antes de poder enviar os dados.

1) **Teste 1 -** Teste para mostrar erro quando não for introduzido todos os parâmetros de registo, para se registar como utilizador (Cidadão) através da aplicação móvel.
    a. Quando não se introduz os parâmetros todos, este envia um alerta a avisar que não preencheu todos os espaços necessários.
    b. Ao pressionar 'OK' este mantém-se na mesma página, mas mantem os dados introduzidos nas caixas de texto.
    c. O *body* enviado de teste: {"nome": "David", "email": "qwerty@hotmail.com", "password": "", "tipo_user": "Cidadão", "ranking_confianca": "0"}.

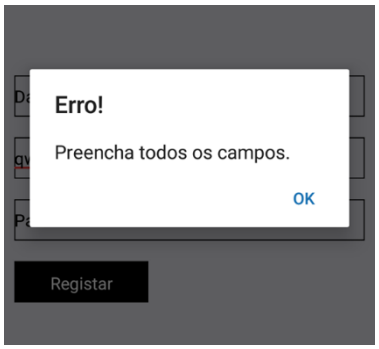


Figura 1 - Erro Preencher Todos Espaços.

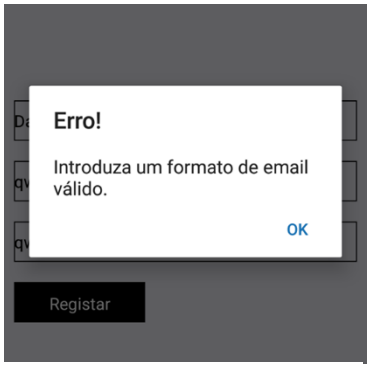


Figura 2 - Erro Introduzir E-mail.

2) **Teste 2 –** Teste para mostrar erro quando for introduzido um parâmetro e-mail no registo com um formato errado no registo do utilizador.

a. Quando se introduz o parâmetro e-mail sem escrever a parte de, por exemplo, '@hotmail.com', este envia um alerta a avisar que o formato do parâmetro enviado não se encontra num formato correto e possível.

b. Ao pressionar 'OK' este mantém-se na mesma página, mas mantem os dados introduzidos nas caixas de texto.

c. O *body* enviado de teste: {"nome": "David", "email": "qwerty", "password": "qwerty", "tipo_user": "Cidadão", "ranking_confianca": "0"}.

3) **Teste 3 –** Teste para mostrar erro quando for introduzido um parâmetro e-mail no registo idêntico a um valor já existente na base de dados.

a. Quando se introduz o parâmetro e-mail já existente na base de dados, este envia um alerta a avisar que esse e-mail já se encontra registado com ele, e daí um cidadão.

b. Ao pressionar 'OK' este mantém-se na mesma página, mas mantem os dados introduzidos nas caixas de texto.

Figura 3 - Erro ao Introduzir E-mail Já Registado na Base de Dados.

c. O *body* enviado de teste: {"nome": "David", "email": "qwerty@hotmail.com", "password": "qwerty", "tipo_user": "Cidadão", "ranking_confianca": "0"}.

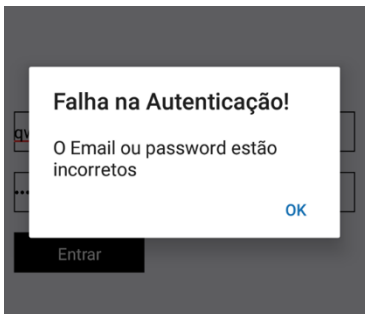


Figura 4 - Erro ao Introduzir E-mail ou Password Durante Login.

4) **Teste 4 –** Teste para mostrar erro, de falha na autenticação, quando for introduzido algum dos parâmetros errados durante o *Login*.

a. Quando se introduz o parâmetro e-mail ou parâmetro password ou ambos errados, em comparação com o que se encontra na base de dados, este envia um alerta a avisar que um desses parâmetros se encontra mal escrito.

b. Ao pressionar 'OK' este mantém-se na mesma página, mas mantem os dados introduzidos nas caixas de texto.

c. O *body* enviado de teste: {"email": "qwerty5@hotmail.com", "password": "qwerty"}.

5) **Teste 5 –** Teste para mostrar erro, de falha no envio do alarme, quando não for introduzido todos os parâmetros para o mesmo, através da aplicação móvel.
    a. Quando não se introduz os parâmetros todos, este envia um alerta a avisar que não preencheu todos os campos necessários, sendo apenas a imagem e a descrição não necessários.
    b. Por exemplo, se os valores do nível de certeza, de perigo de vida e de perigo a edifícios não forem selecionados, estes enviam valor como "NULL", que não é permitido, de acordo com as regras definidas.
    c. Ao pressionar 'OK' este mantém-se na mesma página, mas mantem os dados introduzidos nas caixas preenchidas anteriormente.
    d. O *body* enviado de teste: {"certeza": "NULL", "localizacao": "NULL", "perigo_vida": "NULL", "perigo_edificio": "NULL", "dia": "2018/6/20", "longitude": "-122.36", "latitude": "47.56", "localizacao_nome": "Corroios", "descricao": "FOGO", "id_user": "2"}.

Figura 5 - Erro Preencher Todos os Campos.

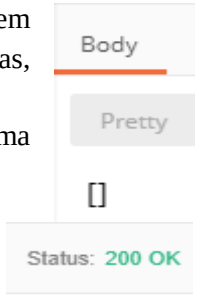
Testes aos *back-end*

Testes para eventual erro:

Uma lista dos testes executados ao *back -end*, através do Postman, de todos os *requests* possíveis, de acordo com a lógica desenvolvida, para retornar os resultados a enviar como resposta, quer estas sejam de erro ou de sucesso.

6) **Teste 6 –** Teste para mostrar resultado quando for introduzido um parâmetro de Id de utilizador para retornar alarmes, através da aplicação móvel, que não existe na base de dados.
    a. Quando se introduz um parâmetro inválido no *url* da operação 'GET' de alarmes, este envia uma lista vazia de objetos.
    b. Por exemplo, se o valor de Id de Utilizador for de '50'.
    c. Não retorna nada, apenas um *status code* a dizer que a operação teve sucesso na pesquisa, o *code* 200.
    d. *Url* de teste:" **http://192.168.1.67:3000/api/Alarme/50** ".
    e. Nota: este erro é impossível de aparecer durante a execução da aplicação móvel, devido ao parâmetro que é passado ser o Id do Utilizador, que é recebido na resposta, ao este dar *login*. Porém, para um Id que não tenha nunca enviado um alarme, será este o resultado de sucesso.

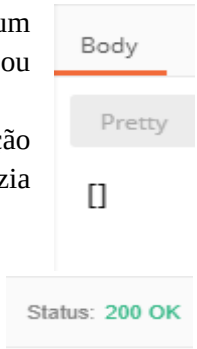
7) **Teste 7 –** Teste para mostrar resultado quando for feito uma pesquisa sem parâmetros para retornar eventos que começaram nos últimos 30 dias, através da aplicação móvel, caso eles não existam na base de dados.
    a. Quando se efetua esta operação 'GET' de eventos, este envia uma lista vazia de objetos.
    b. Não retorna nada, apenas um *status code* a dizer que a operação teve sucesso na pesquisa, o *code* 200.
    c. *Url* de teste:" **http://192.168.1.67:3000/api/Eventos** ".



8) **Teste 8 –** Teste para mostrar resultado quando for feito uma pesquisa sem parâmetros para retornar eventos que não tenham sido classificados, através da aplicação *web* pelos bombeiros, caso todos os eventos tenham sido classificados.
    a. Quando se efetua esta operação 'GET' de eventos, este envia uma lista vazia de objetos.
    b. Não retorna nada, apenas um *status code* a dizer que a operação teve sucesso na pesquisa, o *code* 200.
    c. *Url* de teste:" **http://192.168.1.67:3000/api/Features** ".

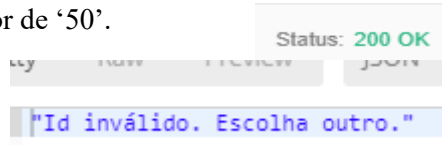
9) **Teste 9 –** Teste para mostrar resultado quando for introduzido um parâmetro de Id de Evento, para retornar as *features* a ele associado, que não existe na base de dados.
    a. Quando se introduz um parâmetro inválido no *url* da operação 'GET' de *features*, este envia uma lista vazia de objetos.
    b. Por exemplo, se o valor de Id de Evento for de '50'.
    c. Não retorna nada, apenas um *status code* a dizer que a operação teve sucesso na pesquisa, o *code* 200.
    d. *Url* de teste:" **http://192.168.1.67:3000/api/Features/50** ".
    e. Nota: este erro é impossível de aparecer durante a execução da aplicação móvel, devido ao parâmetro que é passado ser o Id do Evento, que é recebido na resposta, ao este ser selecionado pelo utilizador. Logo não pode enviar Ids de Eventos que não pode selecionar, obrigando que envie sempre um Id válido. Para qualquer outro Id, estão presentes instâncias.

10) **Teste 10 –** Teste para mostrar resultado quando for introduzido um parâmetro de Id de Evento, para retornar o valor de probabilidade de ser ou não real, a ele associado.
    a. Quando se introduz um parâmetro inválido no *url* da operação 'GET' de probabilidades de evento real, este envia uma lista vazia de objetos.
    b. Por exemplo, se o valor de Id de Evento for de '50'.
    c. Não retorna nada, apenas um *status code* a dizer que a operação teve sucesso na pesquisa, o *code* 200.
    d. *Url* de teste: "**http://192.168.1.67:3000/api/Probabilidade/50**".



e. Nota: este erro é impossível de aparecer durante a execução da aplicação móvel, devido ao parâmetro que é passado ser o Id do Evento, que é recebido na resposta, ao este ser selecionado pelo utilizador. Logo não pode enviar Ids de Eventos que não pode selecionar, obrigando que envie sempre um Id válido. Para qualquer outro Id, estão presentes instâncias.

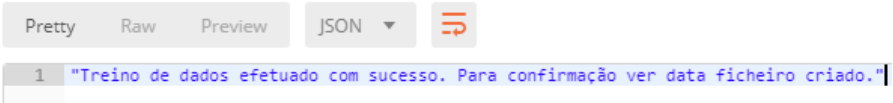
11) **Teste 11 –** Teste para mostrar resultado quando for introduzido um parâmetro de Id de Evento, para retornar o valor de probabilidade de ser um evento ou não real, a ele associado.
    a. Quando se introduz um parâmetro inválido no *url* da operação 'GET' de probabilidades de evento real, este envia uma mensagem de erro de o Id ser inválido.
    b. Por exemplo, se o valor de Id de Evento for de '50'.
    c. Não retorna nada, apenas um *status code* a dizer que a operação teve sucesso na pesquisa, o *code* 200.



    d. *Url* de teste: " **http://localhost:55555/api/predict/50** ".
    e. Nota: este erro é impossível de aparecer durante a execução da lógica da API, devido ao parâmetro que é passado ser o Id do Evento, que é recebido e guardado, durante a atribuição do evento do alarme enviado. Logo não pode enviar Ids de Eventos que não existam na base de dados, obrigando que envie sempre um Id válido.

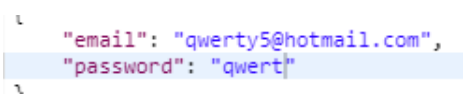
12) **Teste 12 –** Teste para mostrar confirmação que o modelo de *Linear Regression* foi criado/modificado.
    a. Quando se efetua esta operação 'GET' de eventos, este envia uma mensagem para confirmar que o modelo foi treinado, através da modificação da data do ficheiro persistido.
    b. Retorna a mensagem e um *status code*, a confirmar que a operação teve sucesso na pesquisa, o *code* 200.

    c. *Url* de teste: " **http://localhost:55555/api/train** ".

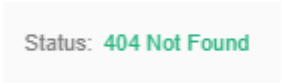


    d. Nota: é impossível aparecer qualquer outro resultado que este, durante a execução da lógica da API, devido aos parâmetros que são utilizados para executar o treino do modelo serem dados préviamente tratados para assegurar que é possível usá-los e os dados provenientes da base de dados sofrerem tratamento, também eles, para ficarem no formato correto. Logo quer na base de dados quer no ficheiro onde guardo informações para treino, mesmo que não se possa recorrer aos dados imediatamente, o seu tratamento é obrigatório e daí ser uma operação sempre bem sucedida.

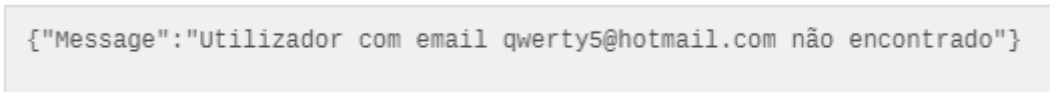
13) **Teste 13 –** Teste para mostrar erro quando dados introduzidos no *login* estão errados, retornando uma mensagem de erro a mostrar tal.
    a. Quando se introduz um dos parâmetros inválidos no *body* da operação 'POST' de *login* de utilizador, este envia mensagem a dizer que esse e-mail não está na base de dados, independentemente de o parâmetro errado ser o e-mail ou a *password*.



    b. Retorna a mensagem e um *status code* a dizer que a operação não encontrou correspondente na base de dados, o *code* 404.

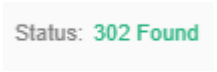


    c. *Url* de teste: " **http://192.168.1.67:3000/api/LogIn** ".

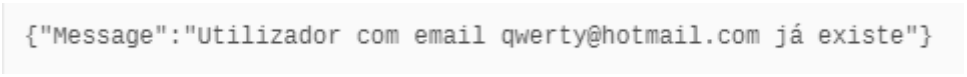


14) **Teste 14 –** Teste para mostrar erro quando dados introduzidos no registar estão errados, retornando uma mensagem de erro a mostrar tal.
    a. Quando se introduz um parâmetro de e-mail inválido no *body* da operação 'POST' de registar um utilizador devido a este já estar presente/registado, este envia uma mensagem a dizer que esse e-mail já se encontra na base de dados.

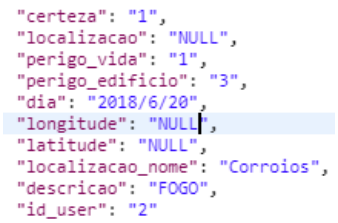
    b. Retorna a mensagem e um *status code* a dizer que a operação não encontrou correspondente na base de dados, o *code* 404.



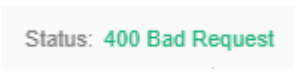
    c. *Url* de teste: " **http://192.168.1.67:3000/api/Registar** ".



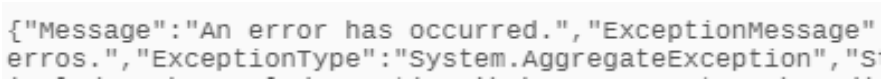
15) **Teste 15 –** Teste para mostrar erro quando dados introduzidos no envio para alarmes estão em formatos errados, retornando uma mensagem de erro a mostrar tal.
    a. Quando se introduz um parâmetro inválido no *body* da operação 'POST' para envio de um alarme devido, por exemplo, a ter um campo com valor nulo, quando este não é permitido. Este envia uma mensagem a dizer que ocorreu um erro.



    b. Retorna a mensagem de erro e um *status code* a dizer que a operação não foi enviada num formato aceite, o *code* 400.

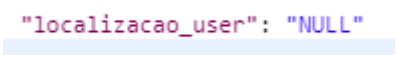


    c. *Url* de teste: " **http://192.168.1.67:3000/api/Alarme** ".



    d. Nota: é apenas possível de aparecer este erro, se utilizador conseguir ser mais rápido que operação para obter a localização do dispositivo móvel.

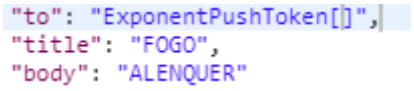
16) **Teste 16 –** Teste para mostrar resultado quando for feito uma pesquisa em que o parâmetro enviado através do *body*, o de localização, não é encontrado na tabela de "**Evento**" durante os últimos 15 dias.
    a. Quando se efetua esta operação 'POST' da localização de eventos, este envia uma lista vazia de objetos.



    b. Não retorna nada, apenas um *status code* a dizer que a operação teve sucesso na pesquisa, o *code* 200.

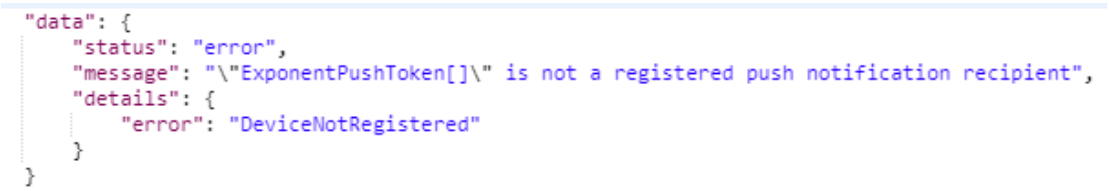
    c. *Url* de teste:" **http://192.168.1.67:3000/api/Eventos** ".

17) **Teste 17 –** Teste para mostrar resultado quando, ao recorrer á utilização de uma API do próprio Expo, para enviar uma notificação para o dispositivo este não enviar o *token* atribuído ao próprio dispositivo para identificação, através do *body*.
    a. Quando se efetua esta operação 'POST' da notificação com respetivo título e texto de mensagem para um *token* nulo, este envia um erro de dispositivo não estar registado.



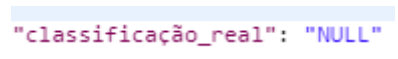
    b. Ainda que retorne uma mensagem de erro, retorna também, porém, um *status code* a dizer que a operação teve sucesso na pesquisa, o *code* 200.

    c. *Url* de teste:" **https://exp.host/--/api/v2/push/send** ".

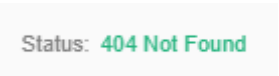


    d. Nota: este erro é impossível de aparecer durante a execução da aplicação móvel, devido ao parâmetro que é passado ser o *token* de identificação do dispositivo, que é atribuido automáticamente. Logo não pode enviar *token* de identificação que não o atribuido, obrigando que envie sempre um *token* válido.

18) **Teste 18 –** Teste para mostrar resultado quando for introduzido um parâmetro de Id de Evento para classificação do mesmo, como sendo um evento real ou não, de um evento para o qual previsões sobre o mesmo, não existam na base de dados.
    a. Apenas quando se introduz um parâmetro inválido no *url* da operação 'POST' de classificação de evento real, sendo aceite um qualquer valor na classificação (*true/false*) ou nenhum valor (nulo), este envia mensagem a informar que não existe previsões para esse Id de evento recebido.



    b. Por exemplo, se o valor de Id de Evento for de '50'.
    c. Retorna a mensagem e um *status code* a dizer que a operação não encontrou correspondente na base de dados, o *code* 404.



d. *Url* de teste: ”**http://192.168.1.67:3000/api/Classificar/50**“.


```
"Message": "Não existe previsões para este evento."
```


e. Nota: este erro é impossível de aparecer durante a execução da aplicação móvel, devido ao parâmetro que é passado ser o Id do Evento, que é recebido na resposta, ao este ser selecionado pelo utilizador. Logo não pode enviar Ids de Eventos que não pode selecionar, obrigando que envie sempre um Id válido.

Testes para eventual sucesso:

19) **Teste 19 –** Teste para mostrar resultado quando for introduzido um parâmetro de Id de Utilizador para retornar alarmes válido, através da aplicação móvel.

a. Quando se introduz um parâmetro válido no *url* da operação ‘GET’ de alarmes, este envia uma lista de objetos, exemplo de objeto em imagem.


```
"ID_Alarme": 2057,
"Certeza": 1,
"Foto": null,
"Tipo_Foto": null,
"Perigo_Vida": 1,
"Perigo_Edificio": 3,
"Localizacao": "NULL",
"Dia": "2018-06-23T00:00:00",
"Localizacao_Nome": "2580 Alenquer, Portugal",
"Latitude": 39.0442142,
"Longitude": -8.987176,
"ID_User": 2,
"ID_Evento": 32,
"Descricao": "Fogo?",
"Evento": null,
"Utilizador": null,
"Imagens": [],
"Resultado_Classificacao": []
```


b. Retorna lista de objetos do tipo Alarme e um *status code* a dizer que a operação teve sucesso na pesquisa, o *code* 200.

c. *Url* de teste:” **http://192.168.1.67:3000/api/Alarme/2** “.

20) **Teste 20 –** Teste para mostrar resultado quando for feito uma pesquisa sem parâmetros para retornar eventos que começaram nos últimos 30 dias, através da aplicação móvel, caso eles existam na base de dados, ordenados por número de alarmes em cada evento.


```
"ID_Evento": 32,
"Localizacao_User": "2580 Alenquer, Portugal",
"Data_Primeiro_A": "2018-06-23T00:00:00",
"Data_Ultimo_A": "2018-07-06T00:00:00",
"Numero_Alarmes": 12,
"Localizacao": "NULL",
"ID_Alarme": 2101,
"Alarme": [],
"Resultado_Probabilistico": [],
"Features": []
```


a. Quando se efetua esta operação ‘GET’ de eventos, este envia uma lista de objetos, como exemplificado na imagem.

b. Retorna lista de objetos do tipo Evento e um *status code* a dizer que a operação teve sucesso na pesquisa, o *code* 200.

c. *Url* de teste:” **http://192.168.1.67:3000/api/Eventos** “.

21) **Teste 21 –** Teste para mostrar resultado quando for feito uma pesquisa sem parâmetros para retornar eventos que não tenham sido classificados, através da aplicação *web* pelos bombeiros.

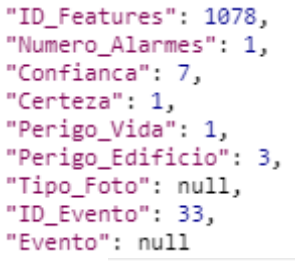

```
"ID_Evento": 33,
"Localizacao_User": "Corroios",
"Data_Primeiro_A": "2018-06-20T00:00:00",
"Data_Ultimo_A": "2018-06-20T00:00:00",
"Numero_Alarmes": 4,
"Localizacao": "NULL",
"ID_Alarme": 2100,
"Alarme": [],
"Resultado_Probabilistico": [],
"Features": []
```


a. Quando se efetua esta operação ‘GET’ de eventos, este envia uma lista de objetos, como exemplificado na imagem.

b. Retorna lista de objetos do tipo Evento e um *status code* a dizer que a operação teve sucesso na pesquisa, o *code* 200.

c. *Url* de teste:" **http://192.168.1.67:3000/api/Eventos** ".

22) **Teste 22 –** Teste para mostrar resultado quando for introduzido um parâmetro de Id de Evento, para retornar as *features* a ele associado.

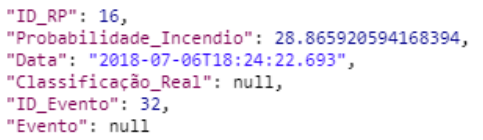


a. Quando se introduz um parâmetro válido no *url* da operação 'GET' de *features*, este envia uma lista de objetos, como exemplificado na imagem.

a. Retorna lista de objetos do tipo Features e um *status code* a dizer que a operação teve sucesso na pesquisa, o *code* 200.

b. *Url* de teste:" **http://192.168.1.67:3000/api/Features/33** ".

23) **Teste 23 –** Teste para mostrar resultado quando for introduzido um parâmetro de Id de Evento, para retornar o valor de probabilidade de ser ou não real, a ele associado.

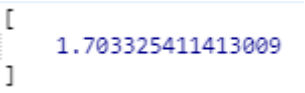


a. Quando se introduz um parâmetro inválido no *url* da operação 'GET' de probabilidades de evento real, este envia uma lista de objetos, como exemplificado na imagem.

b. Retorna lista de objetos do tipo Resultado Probabilístico e um *status code* a dizer que a operação teve sucesso na pesquisa, o *code* 200.

c. *Url* de teste: "**http://192.168.1.67:3000/api/Probabilidade/32**".

24) **Teste 24 –** Teste para mostrar resultado quando for introduzido um parâmetro de Id de Evento, para retornar o valor de probabilidade de ser um evento ou não real, a ele associado.



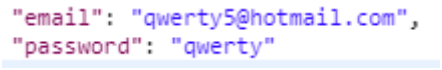
a. Quando se introduz um parâmetro válido no *url* da operação 'GET' de probabilidades de evento real, este uma lista de um único elemento, como exemplificado na imagem.

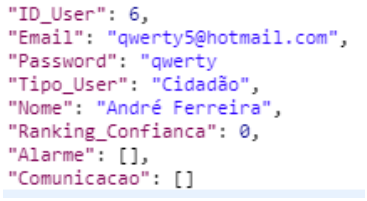
b. Retorna um *array* de um único valor de Resultado Probabilístico e um *status code* a dizer que a operação teve sucesso na pesquisa, o *code* 200.

c. *Url* de teste: " **http://localhost:55555/api/predict/30** ".

25) **Teste 25 –** Teste para mostrar resultado quando dados introduzidos no *login* estão corretos.

a. Quando se introduz todos os parâmetros válidos no *body* na operação 'POST' de *login* de utilizador, este envia mensagem, que contém o Utilizador com que entrou, como exemplificado na imagem de baixo.





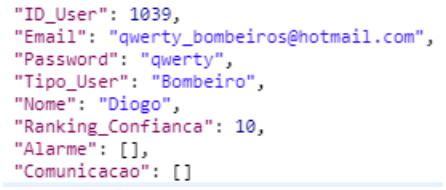
b. Retorna um objeto do tipo de Utilizador e um *status code* a dizer que a operação teve sucesso na pesquisa, o *code* 200.

c. *Url* de teste: " **http://192.168.1.67:3000/api/LogIn** ".

26) **Teste 26 –** Teste para mostrar resultado quando dados introduzidos no registar estão corretos.

a. Quando se introduz todos os parâmetros válidos no *body* da operação 'POST' de registar um utilizador, este envia mensagem, que contém o Utilizador com que entrou, como exemplificado na imagem de baixo.

nome	Diogo
email	qwerty_bombeiros@hotmail.com
password	qwerty
tipo_user	Bombeiro
ranking_confianca	10

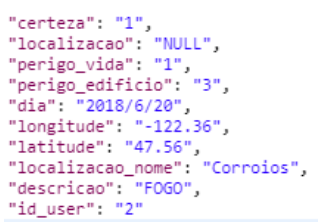


b. Retorna um objeto do tipo de Utilizador e um *status code* a dizer que a operação teve sucesso na pesquisa, o *code* 200.

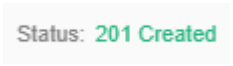
c. *Url* de teste: " **http://192.168.1.67:3000/api/Registar** ".

27) **Teste 27 –** Teste para mostrar resultado quando dados introduzidos no envio para alarmes estão no formato correto.

a. Quando se introduz todos os parâmetros válidos no *body* da operação 'POST' para envio de um alarme, este envia uma mensagem que contém apenas o *status code*, como exemplificado na imagem do lado.

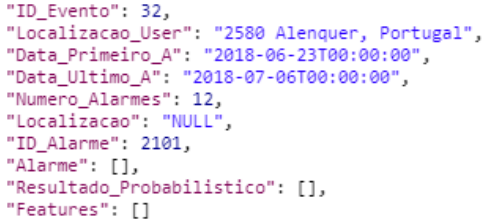


b. Retorna apenas um *status code* a dizer que foi criado um objeto na base de dados, o *code* 201.

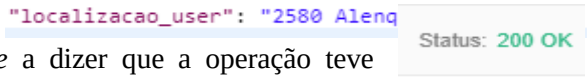


c. *Url* de teste: " **http://192.168.1.67:3000/api/Alarme** ".

28) **Teste 28 –** Teste para mostrar resultado quando for feito uma pesquisa em que o parâmetro enviado através do *body*, o de localização, é encontrado na tabela de "**Evento**" durante os últimos 15 dias.

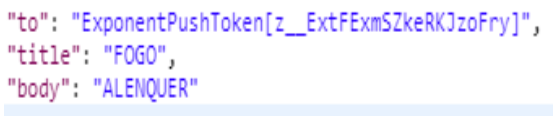


a. Quando se efetua esta operação 'POST' da localização de eventos, este envia uma lista de objetos, como exemplificado na imagem.
b. Retorna lista de objetos do tipo Evento e um *status code* a dizer que a operação teve sucesso na pesquisa, o *code* 200.



c. *Url* de teste:" **http://192.168.1.67:3000/api/Eventos** ".

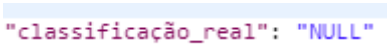
29) **Teste 29 –** Teste para mostrar resultado quando, ao recorrer á utilização de uma API do próprio Expo, para enviar uma notificação para o dispositivo, este enviar o *token* atribuído ao próprio dispositivo para identificação, através do *body*.
a. Quando se efetua esta operação 'POST' da notificação com respetivo título e texto de mensagem para um certo *token*, este envia uma mensagem a informar se conseguiu enviar notificação.



b. Retorna um dado a confirmar envio de notificação e um *status code* a dizer que a operação teve sucesso na pesquisa, o *code* 200.

c. *Url* de teste:" **https://exp.host/--/api/v2/push/send** ".

30) **Teste 30 –** Teste para mostrar resultado quando for introduzido um parâmetro de Id de Evento para classificação do mesmo, como sendo um evento real ou não, de um evento para o qual previsões sobre o mesmo existam na base de dados.
a. Apenas quando se introduz um parâmetro válido no *url* da operação 'POST' de classificação de evento real, sendo aceite um qualquer valor na classificação (*true/false*) ou nenhum valor (nulo), este envia apenas um *status code* a informar se teve sucesso.



b. Retorna lista de objetos do tipo Evento e um *status code* a dizer que a operação teve sucesso na pesquisa, o *code* 200.

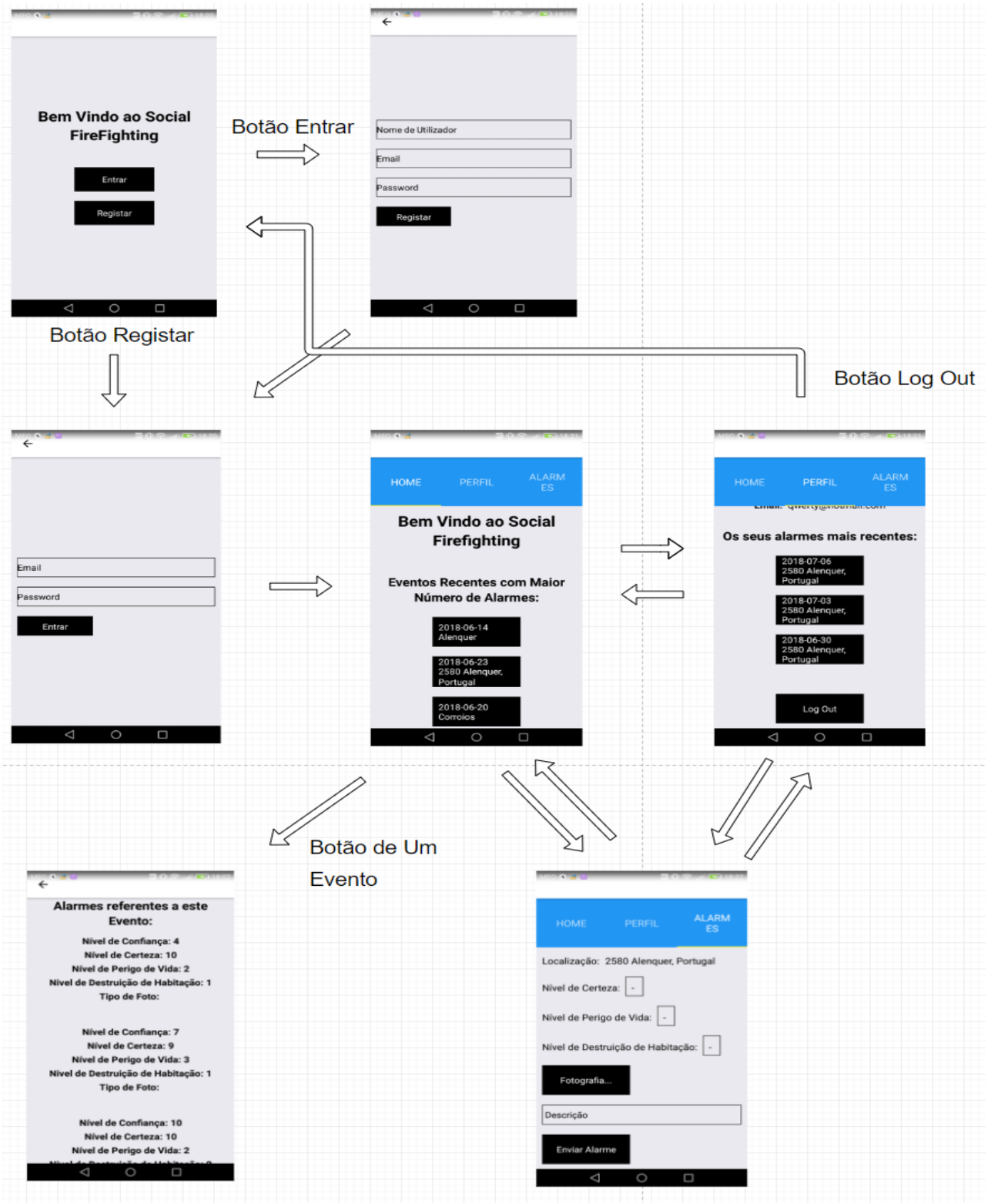
c. *Url* de teste: "**http://192.168.1.67:3000/api/Classificar/30**".

Resultados

Os resultados finais obtidos, demonstrados através das duas aplicações, móvel e *web*, em execução.

Resultado Final da Aplicação Móvel

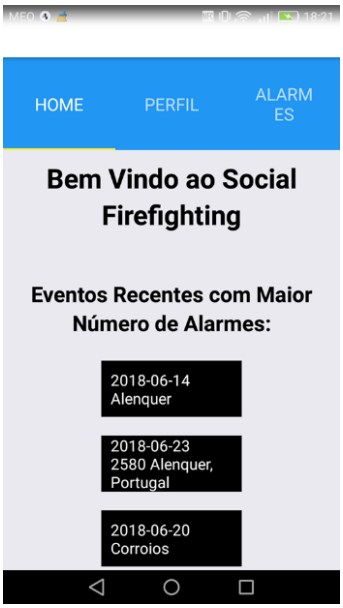
- **Flowchart**:



- Resultados obtidos ao entrar (dar *login*) na aplicação:

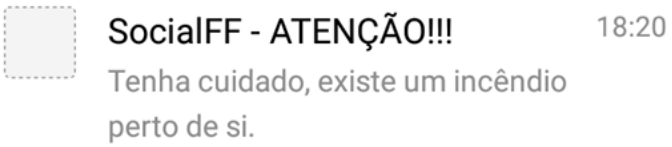
- Resultados obtidos para demonstração de Eventos:

Ao entrar na aplicação, seguindo o *flowchart* definido acima, esta deve passar para página 'HOME' e, automaticamente durante a montagem da página, executar um *fetch* (uma operação que cria e envia um *request* á API desenvolvida através do *url* definido) que retorna os últimos 5 eventos com maior número de alarmes, dos últimos 30 dias. Como demonstrado abaixo:



- Resultados obtidos caso exista evento na sua localização:

Ao entrar, esta deve também confirmar se há ou não um evento na localização do utilizador, através de um *fetch* previamente definido para tal e executado automaticamente durante a montagem da página, apresentando como resultado uma notificação caso este exista. Um exemplo dessa notificação é:

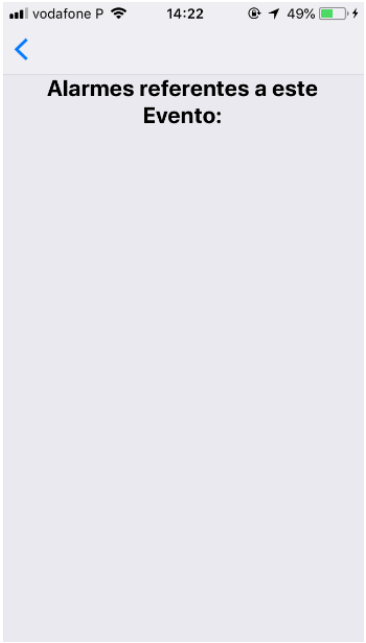


- Resultado obtido ao selecionar a *Tab* 'PERFIL' na aplicação:

Ao selecionar a *Tab* 'PERFIL', a aplicação deve automaticamente, durante a montagem da página, executar um *fetch* que retorne os últimos 3 eventos enviados pelo utilizador. Deve também criar o perfil do utilizador que deu *login*, demonstrado através da aparência do nome e do e-mail com que este entrou na aplicação. Como demonstrado abaixo:

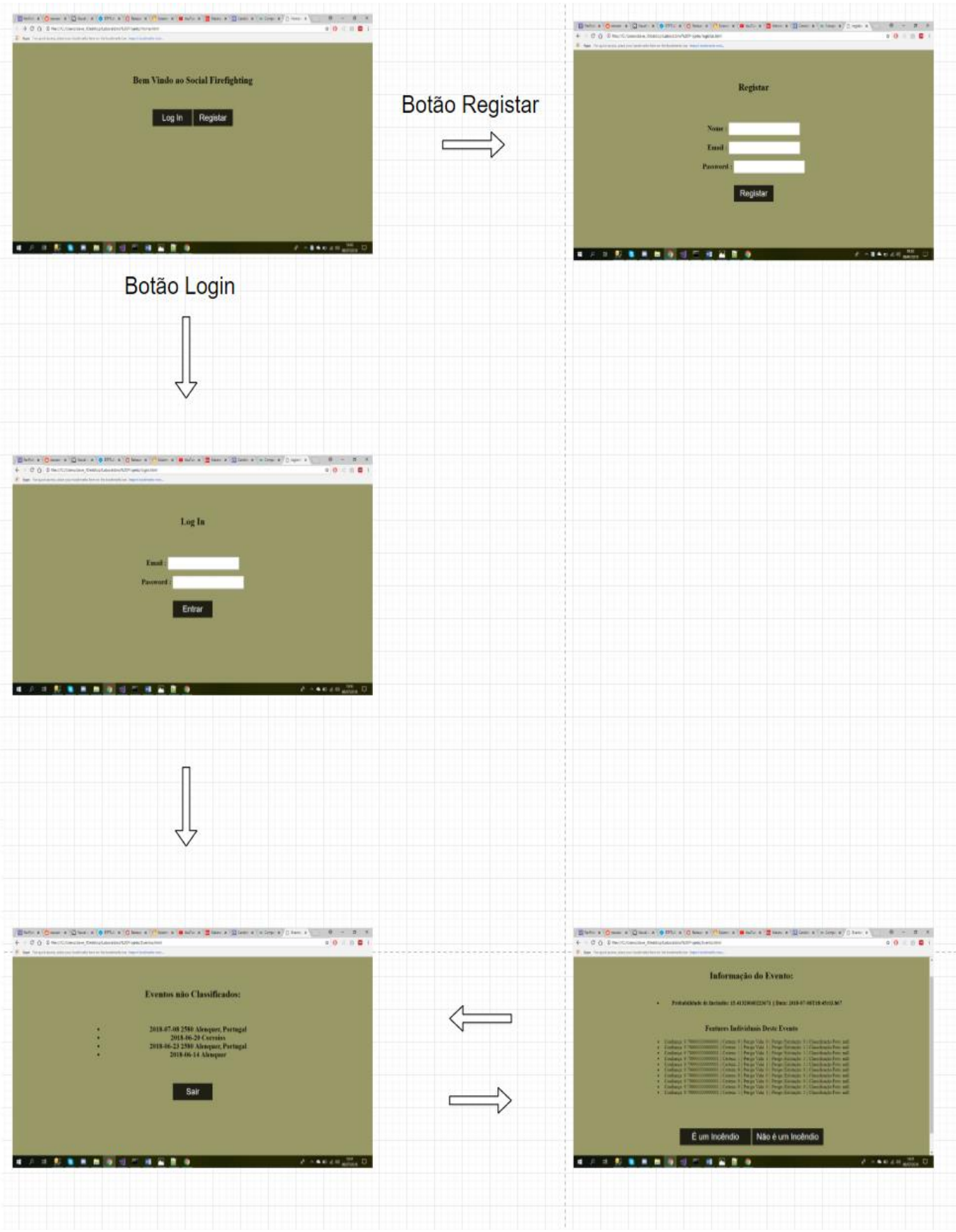
- Resultado obtido ao selecionar um dos eventos da *Tab* 'HOME' na aplicação:

Ao selecionar um dos eventos da *Tab* 'HOME', a aplicação deve automaticamente, durante a montagem da página, executar um *fetch* que retorne todos as *features* enviadas pelos utilizadores sobre o evento selecionado. Como demonstrado abaixo:



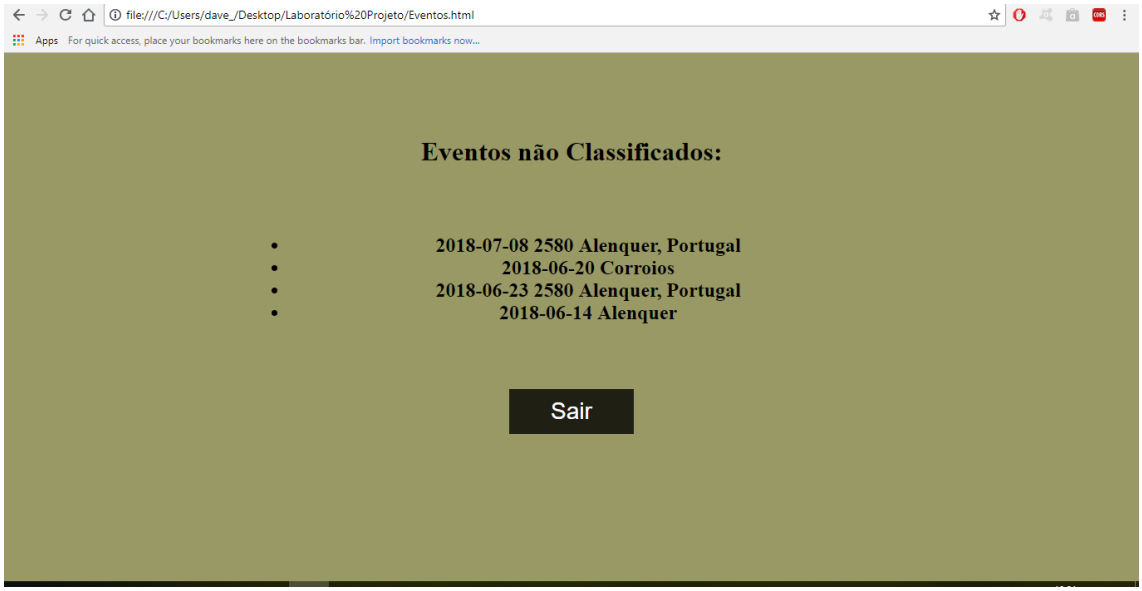
Resultado Final da Aplicação *Web*

- **Flowchart:**



- **Resultados ao bombeiro dar *login*:**

Ao dar login na aplicação *web* como bombeiro, este deverá mostrar todos os eventos por classificar e apenas eles, sendo que se o bombeiro classificar um deles, então esse será retirado da lista de apresentados. Apresenta também um botão para dar *logout* da aplicação.



- **Resultados ao clicar num evento:**

Ao clicar num dos eventos da lista apresentada anterior este deve apresentar a lista de *features*, criadas com base nos alarmes enviados pelos cidadãos, e a última previsão, com respetiva data em que foi feita, da probabilidade de ser ou não um evento real.

  - **Resultado da probabilidade de ser evento real:**

No exemplo presente na imagem abaixo, nota-se que a probabilidade de o evento ser um incêndio real é de, mais ou menos, 15.41. Este resultado explica-se pelo facto de, segundo as regras com que se criou o ficheiro *.csv*, que contém maior carga de eventos classificados e que foi utilizado para executar o treino do modelo de *Linear Regression* (logo com maior peso na aprendizagem), as regras para ser considerado sucesso a que este evento corresponde, não são nenhumas das 5 *features* disponíveis, sendo a mais perto o número de alarme maior que 10, com a presença exata de 10 alarmes.

Assim, torna-se fácil de calcular a probabilidade que este vai apresentar neste evento. Tendo 5 *features*, para cada uma delas a que evento corresponde obtém-se a probabilidade de 1/5 de 100%, logo 20%. Como este evento corresponde apenas a uma *feature*, a de uma média de confiança maior que 5, obtém uma percentagem imediata alta. Porém este encontra-se também quase a corresponder a outra *feature*, a de atingir mais que 10 alarmes, tendo atingindo já 10. Porém, todos os outros valores encontram-se com médias claramente negligentes, o que baixa a probabilidade de o evento ser um incêndio real.

Logo pelo contrabalanço encontrado pelo modelo, este atribui-lhe uma probabilidade de pelo menos 20% por atingir pelo menos uma das 5 *features*, ao qual vai retirar importância pelas restantes médias atingirem valores que apontam claramente para que não seja um evento, fazendo com que retire maior parte dessa probabilidade, mas recebeu 10 alarmes para este evento, quase

o suficiente para atingir uma outra *feature*, daí dar-lhe importância na mesma. Todo este caminho faz com que evento fique com uma média de probabilidade de 15.

- **Resultado ao clicar em classificar evento:**

Ao clicar num dos dois botões para classificar como sendo ou não um evento, este deve automaticamente modificar o valor da confiança dos utilizadores que enviaram alarmes para o evento classificado. Como resultado deste caso, é possível verificar-se na imagem acima, que o valor da confiança do utilizador que enviou alarme, sofreu modificações manuais e não por ter sido introduzido na base de dados com um valor de 9.7. Caso assim fosse, este mostraria apenas 9.7 sem mais nada á frente. O contrário se passa neste caso, onde mostra uma data de zeros á direita do 9.7, até este ser arredondado.

Este evento vai também a ser considerado e utilizado quando, no futuro, se executar a operação de treino do modelo de *Linear Regression*.

Conclusão

Neste projeto correspondemos a quase todos os objetivos a que nos comprometemos realizar, com o único percalço sendo o envio e classificação das imagens tiradas pelos cidadãos sobre os fogos e enviados com o alarme.

Assim, desenvolvemos uma aplicação móvel simples capaz de apresentar informação sobre os eventos e alarmes e enviar alarmes para uma REST API a partir da qual faremos o tratamento dos dados recebidos e guardaremos os dados finais na base de dados desenvolvida para tal.

Desenvolvemos também uma aplicação *web* para apresentação dos alarmes e eventos aos bombeiros no centro de comando, com uma respetiva classificação de probabilidade de ser um evento real, permitindo a sua classificação.

- ## Considerações futuras:

Algumas considerações futuras a ter para melhorar este projeto passa pela implementação do envio e apresentação de imagens, pelos cidadãos para os bombeiros, com a respetiva classificação das imagens como sendo sobre um fogo ou não.

Implementar a possibilidade de mostrar o alarme completo, com a respetiva ou não imagem enviada, ao cidadão na aplicação móvel, ao selecionar um dos últimos 3 alarmes que este enviou.

Implementar em ambas as aplicações uma zona de chat de modo ao bombeiro poder comunicar diretamente com o cidadão que enviou um certo alarme.

Melhorar o *dataset* através de dados reais, em vez de dados fabricados pelo grupo, fornecidos preferencialmente pela Organização de Bombeiros.

Permitir que utilizador modifique a sua informação no respetivo perfil de utilizador, acrescentando a possibilidade de adicionar uma foto de perfil, com outros parâmetros também de importância, por exemplo o número de telemóvel.

Ao enviar um alarme, recarregar a informação dos últimos alarmes enviados.

Possibilitar correr automaticamente o método para treinar o modelo de *Linear Regression*, em certos períodos de tempo.

Melhorar o aspeto da aplicação *web* e da aplicação móvel, e as suas respetivas páginas e *tabs*.

Melhorar relatório de alarme a ser enviado, adicionando novos campos de informação a enviar, incluindo um campo para envio da localização escolhida por utilizador, a partir do GOOGLE Maps, permitindo que este envie alarmes para incêndios fora do seu alcance.

Permitir que bombeiros classifiquem um incêndio como tendo sido apagado, permitindo que alarmes enviados logo de seguida nessa localização sejam classificados como um novo evento em vez de ser o mesmo.

Permitir que, em vez de utilizador escolher uma foto a enviar como agora implementado, este retirar uma foto diretamente.

Agradecimentos

Agradecemos ao professor Nuno Brás, da Universidade Autónoma de Lisboa, pela orientação e ajuda para o desenvolvimento do projeto ao longo deste último semestre.

Agradecemos a Filipe Caçador, Mestre de Engenharia Informática, pela ajuda inicial para o desenvolvimento da API, orientando o caminho para escolha de Software de desenvolvimento, o Visual Studio 2017 e o Microsoft SQL Server.

Listagem de Software

- **Notepad++**:
  - Um editor de *source code* e bloco de notas que providencia suporte a várias linguagens, que corre no ambiente virtual de Microsoft Windows, governado pela licença GPL (*Gnu General Public License*), que requer que software derivado deste seja licenciado sob esta licença.
  - Baseado no componente Scintilla, este é escrito em C++ e recorre á utilização de Windows 32 API e STL, que permite assegurar maior velocidade de execução e menor dimensão do programa **[25]**.
  - Em relação ao projeto desenvolvido, recorreu-se á sua utilização para escrita do código das aplicações de *front-end*: a aplicação *cross-platform*, para o dispositivo móvel em React Native, e para a aplicação *web* de apresentação ao bombeiro no comando.
- **Chrome**:
  - Um *web browser* desenvolvido pelo GOOGLE com o objetivo de ser rápido em todas as suas operações, sendo a sua utilização intuitiva e simples e procura ser o mais seguro possível durante a sua utilização.
  - Este disponibiliza mais de 150,000 extensões durante a navegação na *web* **[26]**.
  - Em relação ao projeto desenvolvido, recorreu-se á sua utilização para a execução da aplicação *web* de *front-end* de apresentação ao bombeiro, tendo recorrido a uma das suas extensões para permitir envio de *requests* e leitura das respostas dadas pela API desenvolvida pelo grupo.
  - Extensão Allow-Control-Allow-Origin: * :
    - Permite CORS (Cross-Origin Resource Sharing), um mecanismo que permite à linguagem de programação JavaScript numa página *web* enviar *Requests* a outro domínio, que não o original onde esta página se encontra **[27]**.
- **Expo**:
  - É um *toolchain* de código aberto e grátis construído á volta da linguagem de programação React Native para a criação de projetos nativos em IOS e Android, assim sendo projetos *Cross-Platform*, recorrendo às linguagens de programação JavaScript e React para a construção de APPs **[28]**.
  - Para tal, recorre a Expo SDK, uma biblioteca em Native e JS que permite acesso às funcionalidades do sistema do dispositivo em que está instalado (exemplos pertinentes ao nosso projeto: acesso á câmara e localização corrente do dispositivo), que permite correr os projetos em questão em qualquer ambiente que contenha a Expo SDK **[29]**.

- Em relação ao projeto desenvolvido, recorreu-se á sua utilização para a execução da aplicação *cross-platform* de *front-end* no dispositivo móvel e para a utilização da biblioteca Expo SDK para pedir permissões de acesso, como dito anteriormente, às imagens de câmara e á localização corrente do dispositivo para criação de alarmes e divisão em eventos.

- **Node.js**:
  - É um JavaScript dirigido a eventos assíncronos desenhado para permitir a construção de aplicações de rede escaláveis de alta performance **[30]**.
  - Em relação ao projeto desenvolvido, recorreu-se á sua utilização devida á necessidade da sua instalação para permitir a utilização do Expo.
- **Visual Studio 2017**:
  - Um IDE (*Integrated Development Environment* ou Ambiente de Desenvolvimento Integrado) da Microsoft que reúne as ferramentas de apoio ao desenvolvimento de software e que suporta linguagens de programação como Visual Basic, .NET Framework, C, C#, C++, J# e outras compatíveis a plataforma ASP.NET, utilizada para desenvolvimento de produtos na web, como *websites*, aplicações *web*, serviços *web* e aplicações móveis **[31]**.
  - Em relação ao projeto desenvolvido, recorreu-se á sua utilização para criação de duas APIs: uma .NET REST API, á qual se recorreu às linguagens de programação compatíveis com ASP.NET; e uma Flask REST API, á qual se recorreu á linguagem Python. Para tal, necessitou-se a instalação dos respetivos pacotes.
  - Recorreu-se ainda á criação de um ficheiro de Entity Framework, conectado á base de dados criada, responsável pela criação dos modelos de dados que esta permite inserir.
  - Finalmente, recorreu-se á utilização de LINQ Queries para executar perguntas á base de dados e recolher informação da mesma, através do Entity Framework.
  - Assim, foi utilizado como servidor que permite fazer as conexões entre o *front-end* e o *back-end* (que inclui a própria API) do projeto.
- **Microsoft SQL Server Management Studio 17**:
  - Um ambiente integrado para gerir infraestruturas de Microdoft SQL, de SQL Server ou Azure SQL Data e Database, desenvolvido pela Microsoft. Providencia ferramentas para configurar, monitorizar e administrar instâncias de SQL **[32]**.
  - Em relação ao projeto desenvolvido, recorreu-se á sua utilização para gerir e monitorizar a base de dados desenvolvida em Microsoft SQL Server.
  - Assim, este foi utilizado para gerir e monitorizar o *back-end* (a base de dados) do projeto.
- **Microsoft SQL Server**:
  - Um SGBD (Sistema de Gerenciamento de Banco de Dados Relacional ou *Data Base Management System*) desenvolvido pela Microsoft. Como uma *Data Base* é um produto de software com a função de armazenamento e recuperação dos dados solicitados por outras aplicações, quer no mesmo computador quer através de uma rede.
  - Para tal utiliza linguagens de consulta Transact-SQL (T-SQL) e ANSI SQL **[33]**.
  - Em relação ao projeto desenvolvido, recorreu-se á sua utilização para desenvolver a base de dados, incluindo as tabelas e inserção de dados básicos para testes.

    - Assim, este foi utilizado para desenvolver o *back-end* (a base de dados) do projeto.
- **Postman**:
    - Um ADE (API Development Environment), ou seja, uma plataforma que suporta e melhora o desenvolvimento das APIs, simplificando o processo de desenvolvimento, centralizando a API da organização e melhorando a colaboração com as APIs através da organização.
    - Possibilita a opção de documentar, testar (através da execução de *Requests* á *web* API), monitorização, *debug* e publicação de uma API **[34]**.
    - Em relação ao projeto desenvolvido, recorreu-se á sua utilização para desenvolvimento das duas APIs desenvolvidas, tendo sido utilizado para executar tarefas de *debug* e teste, através do envio de *requests* às duas APIS, tendo assim permitido com maior facilidade criar os mesmos durante a sua implementação nos respetivos *front-end*.
    - Assim, este foi utilizado para permitir fazer testes ao *back-end* (API + base de dados) do projeto.
- **Anaconda 3**:
    - Um software grátis e *open source* para distribuição de linguagens de programação Python e R, muito utilizado para aplicações que envolvem *Data Science* e *Machine Learning*, comoprocessamento de dados de larga escala, análise preditiva e computação científica.
    - Tem o objetivo de simplificar a gestão dos pacotes das respetivas linguagens e a sua execução **[35]**.
    - Em relação ao projeto desenvolvido, recorreu-se á sua utilização para desenvolvimento de código e teste do mesmo no seu ambiente de desenvolvimento, o Jupyter, para executar tarefas de *machine learning*, com o objetivo de criar um modelo básico para determinar a probabilidade de certo evento ser um incêndio real.
    - Após o seu desenvolvimento, o código foi transportado e implementado no IDE Visual Studio para criação da REST API em Flask (desenvolvida em Python).
    - Para tal, recorreu-se ao modelo de Regressão Linear, sendo treinado com os dados iniciais da tabela previamente classificados e retornando o valor da previsão para os dados de certo evento.
    - Assim, este foi utilizado para criação de opções de *back-end* (Flask API) do projeto.

Bibliografia

[1] - https://pt.wikipedia.org/wiki/Inc%C3%AAndio

[2] - https://www.cmjornal.pt/portugal/detalhe/pior-dia-do-ano-de-incendios-diz-adjunta-da-protecao-civil

[3] - https://becode.com.br/back-end-front-end-full-stack/

[4] - https://www.kwan.pt/pt/blog/back-end-dev

[5] - https://canaltech.com.br/software/o-que-e-api/

[6] - https://www.9lessons.info/2012/05/create-restful-services-api-in-php.html

[7] - https://tutorialedge.net/general/what-is-a-rest-api/

[8] - https://pt.wikipedia.org/wiki/REST

[9] - http://www.scriptcaseblog.com.br/request-e-response/

[10] - https://www.tutorialspoint.com/http/index.htm

[11] - https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status

[12] - http://www.ccg.pt/machine-learning-o-que-e/

[13] - https://www.sas.com/pt_br/insights/analytics/machine-learning.html

[14] - https://machinelearningmastery.com/linear-regression-for-machine-learning/

[15] - https://pt.wikipedia.org/wiki/Regress%C3%A3o_linear

[16] - http://scikit-learn.org/stable/

[17] - http://scikit-learn.org/stable/modules/model_persistence.html

[18] - https://pt.wikipedia.org/wiki/ADO.NET_Entity_Framework

[19] - https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/connection-string-syntax

[20] - https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/introduction-to-linq-queries

[21] - http://www.organicadigital.com/seeds/o-que-e-react-native/

[22] - https://expo.io/

[23] - https://blog.mxcursos.com/front-end-ou-back-end-entenda-as-diferencas-e-descubra-o-seu-perfil/

[24] - https://pt.wikipedia.org/wiki/Automa%C3%A7%C3%A3o_de_teste

[25] - https://notepad-plus-plus.org/

[26] - https://www.google.com/chrome/

[27] – https://chrome.google.com/webstore/detail/allow-control-allow-origi/nlfbmbojpeacfghkpbjhddihlkkiljbi

[28] - https://expo.io/

[29] - https://docs.expo.io/versions/latest/

[30] - https://nodejs.org/en/about/

[31] - https://pt.wikipedia.org/wiki/Microsoft_Visual_Studio

[32] - https://docs.microsoft.com/en-us/sql/ssms/download-sql-server-management-studio-ssms?view=sql-server-2017

[33] - https://pt.wikipedia.org/wiki/Microsoft_SQL_Server

[34] – https://www.getpostman.com/products

[35] - https://en.wikipedia.org/wiki/Anaconda_(Python_distribution)

https://facebook.github.io/react-native/docs/getting-started.html

https://reactnavigation.org/

https://github.com/

https://www.npmjs.com/

https://codeburst.io/integrating-react-native-apps-with-back-end-code-using-fetch-api-8aeb83dfb428

https://hackernoon.com/react-native-basics-geolocation-adf3c0d10112

https://medium.com/

https://docs.microsoft.com/en-us/

https://stackoverflow.com/questions/

https://www.tutorialspoint.com/ms_sql_server/index.htm

https://www.w3schools.com/sql/

https://docs.Python.org/

https://machinelearningmastery.com/

https://www.wintellect.com/creating-machine-learning-web-api-flask/

https://www.wintellect.com/creating-a-simple-linear-regression-machine-learning-model-with-scikit-learn/

https://auth0.com/blog/developing-restful-apis-with-Python-and-flask/

https://wiki.Python.org/moin/

http://scikit-learn.org/

https://medium.com/@alexkiefer_23564/easily-setup-push-notifications-with-expo-for-react-native-apps-be50668832e2