ISEL – Instituto Superior de Engenharia de Lisboa
ADEETC – Área Departamental de Engenharia de Electrónica e Telecomunicações e de Computadores

LEIM
Licenciatura em Engenharia Informática e Multimédia
Unidade Curricular de Projeto

Life Guardian System (LGS)

Francisco Reis (44616)

Gabriel Diaz (45133)

Orientador(es)

Professor Doutor Paulo Trigo

Engenheiro Bruno Coelho

Setembro, 2020

Resumo

Todos os momentos de crise imprevista levam a momentos de reflexão e introspecção. Será que efectivamente não havia maneira de prever essa crise? Não existiram indicadores de que tal poderia acontecer? Infelizmente a condição humana só nos faz colocar estas questões após os acontecimentos e nunca antes.

Uma das áreas em que isto ocorre de forma mais imprevisível (além dos desastres naturais) é na saúde. Só em Portugal nomes como Francisco Lázaro e Miklós Fehér relembram-nos da fragilidade inerente da condição humana e que, por vezes, os acontecimentos mais improváveis são a realidade.

Casos destes levam-nos a questionar se, de facto, não haveriam indicadores ou sinais que pudessem conduzir a este desfecho muito antes de ele se pronunciar.

Da tentativa de criar uma ferramenta que auxilie na prevenção de situações semelhantes surge este projecto. A implementação de um serviço que, através da recolha e processamento de informação sensorial, avalia os seus parâmetros vitais. Caso estes se desviem de uma certa gama de valores pré-definidos é notificado o utilizador e é dada a possibilidade de contactar, em seu nome, alguém na lista de contactos de emergência, previamente definida, para que o possam tentar socorrer.

Este é estruturado de forma a suportar qualquer tipo de indicador tendo apenas de se proceder à implementação dos algoritmos necessários ao seu processamento e é escalável na medida em que basta replicar o que já está feito em mais equipamentos para comportar um maior volume de utilizadores.

Agradecimentos

A elaboração do presente trabalho não seria possível sem o apoio de alguns intervenientes. Assim sendo, pretendemos agradecer a todos os que apoiaram e contribuíram para a realização e concretização deste projecto final.

Deste modo agradecemos:

- A todos os familiares e amigos que nos deram forças para continuar especialmente nas alturas em que a motivação escasseava e as duvidas surgiam.
- Ao Eng.Bruno Coelho por nos ter apoiado e guiado ao longo de todo o projecto.
- Ao Eng.Paulo Trigo pelo seu apoio na estruturação e apresentação de documentos em todas as apresentações que foram feitas no decorrer do semestre.
- Finalmente a todos os docentes que contribuíram para a nossa formação ao longo da licenciatura, por todos os conhecimentos, dedicação e contributo para o nosso crescimento pessoal e educacional.

A todos, muito obrigado...

Índice

Lista de Figuras

Lista de Tabelas

Siglas

API Application Programable Interface.

CRUD Create,Read,Update,Delete.

FQN Fully Qualified Name.

HTTP Hyper Text Transfer Protocol.

JMS Java Messaging Service.

JSON JavaScript Object Notation.

JWT JSON Web Token.

KPI Key Performance Indicators.

LGS Life Guardian System.

MVC Model View Controller.

POM Project Object Model.

RAML RESTful API Modeling Language.

REST Representational State Transfer.

SAAS Software as a Service.

SMS Simple Message Service.

SMTP Simple Mail Transfer Protocolo.

SOAP Simple Object Access Protocol.

SQL Structured Query Language.

UML Unified Modeling Language.

URL Uniform Resource Locator.

UUID Universally Unique Identifier.

WHMD Wearable Health Monitoring Devices.

WSDL Web Services Description Language.

XML Extensible Markup Language.

Capítulo 1

Introdução

Um dos tópicos actuais que preocupa as sociedades e que muitas vezes não tem respostas exactas é o da saúde, mesmo pessoas com preparação física acima da média que, à partida, não teriam razões para suspeitar de problemas de saúde podem subitamente ver a sua higidez comprometida.

Associados a estes podemos também falar dos doentes crónicos que precisam de monitorização mais frequente da sua condição física e o cidadão comum que por vezes só se apresenta num gabinete médico algum tempo após se sentir mal.

Surge deste tópico em especifico a necessidade de criar sistemas de monitorização que possam ser utilizados por toda e qualquer pessoa seja qual for a sua localização e que permitam uma visualização destes dados ulteriormente.

Este trabalho pretende implementar uma API na forma de um Software as a Service, que é a interligação entre um sistema de *back-end* que disponibiliza um conjunto de serviços por via de uma API RESTfull a dispositivos capazes de enviarem e receberem pedidos HTTP.

Estes serviços, além das funcionalidades CRUD do sistema para manipular dados sobre utilizadores e informações variadas, disponibiliza um serviço de processamento de dados sensoriais na forma de verificação de desvio de uma certa gama de valores aos seus utilizadores com o intuito de os notificar.

Capítulo 2

Trabalho Relacionado

Efectivamente existe um conjunto de sistemas destinados ao mundo da medicina que permite aos pacientes serem monitorizados em tempo real a partir de casa com recurso a sistemas de monitorização wireless (Wearable Health Monitoring Devices (WHMD)). Contudo, na maioria estes sistemas são apenas utilizados em pacientes crónicos que apresentem um risco constante de deterioração do estado de saúde. Na revista [MDtechreview, 2019] existe um artigo sobre este tema que apresenta uma lista das 10 melhores tecnologias de monitorização do estado de saúde à altura de escrito.

Estas tecnologias visam monitorizar doentes crónicos remotamente de forma a que não ocupem um quarto de hospital apenas para questões de monitorização e apresentam uma especialização num tipo especifico de sensor focando-se em garantir a precisão e veracidade da informação que recebem. Porém, com base no artigo [Yokota, 2020] e na sua avaliação sobre o valor da informação conclui-se que a avaliação do estado de saúde de uma pessoa será tanto mais fiável quanto o numero de parâmetros biométricos utilizados para essa avaliação.

Em concordância com esta ideologia surge o sistema apresentado pela empresa [VoCare, 2019] que implementou um sistema extremamente semelhante ao que se pretende implementar neste projecto. Um sistema portátil que contem múltiplos sensores que o utilizador transporta para ser constantemente monitorizado.

O factor de diferenciação entre este projecto e o que se pretende implementar incide no sistema de notificações com base em algoritmos de avaliação de parâmetros vitais para a população em geral e na separação dos dispositivos sensoriais do aparelho que comunica com o *back-end*.

Até a data os sistemas de gestão de risco (especialmente na área da saúde) são principalmente reactivos, focando-se em identificar possíveis situações de risco e mitigarem os resultados de uma situação desse género ocorrer. Isto advêm da incapacidade do ser humano prever o resultado exacto de um acontecimento como abordado na saga [Taleb, 2016] de *Nassim Nicholas Taleb* em livros como [Taleb, 2012b] e [Taleb, 2012a] que falam sobre acontecimentos improváveis e o seu impacto.

Capítulo 3

Modelo Proposto

O desenvolvimento deste sistema tem por objectivo disponibilizar, além das funcionalidades CRUD para manipular dados sobre utilizadores e informações variadas, uma funcionalidade que, ao receber dados biométricos sobre um dado utilizador, compara-os contra um conjunto de escalas pré-definidas de valores considerados padrão para esses dados e processa-os com recurso a um algoritmo para identificar se é necessário enviar notificações a uma lista de contactos pré-definidos pelo utilizador para esse facto.

Conceptualmente definiu-se que:

1. Cada sensor mede um tipo de sinal biométrico
2. Um dispositivo comunica com um ou mais sensores e com o sistema.

Espera-se que chegue ao sistema, sobre a forma de uma mensagem, a informação sensorial de um dado utilizador, com esta informação o sistema estrutura uma nova mensagem que contem informação sobre como se comparam esses dados aos parâmetros guardados em memoria e expedita-a de volta ao utilizador.

De forma a facilitar a visualização do sistema no seu todo apresenta-se em seguida uma figura representativa do mesmo:

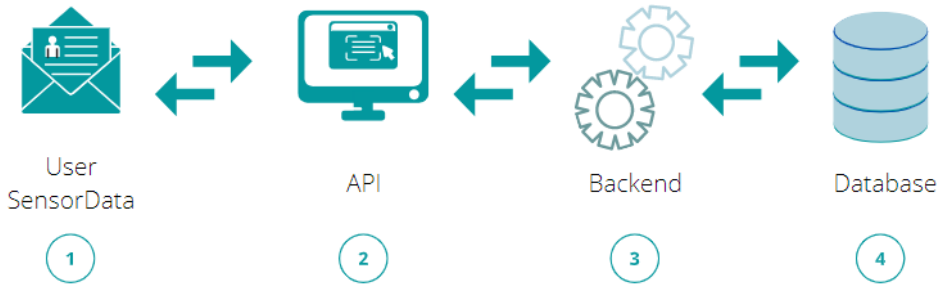

Figura 3.1: Esquema Ilustrativo do modelo

Neste pode se visualizar os vários elementos constituintes do sistema

1. As mensagens trocadas entre o sistema e o utilizador
2. A API disponibilizada para este efeito
3. O sistema em si que trata do processamento dos dados
4. A base de dados que preserva todas as informações necessárias

3.1 Requisitos

Antes de implementar o sistema em questão foi feito um levantamento de requisitos sobre quais as funcionalidades que seriam implementadas, quais poderiam ser implementadas e quais os dados que seriam necessários entregar ao sistema para que este pudesse realizar as operações pretendidas.

Para tal, primeiro foi preciso definir, qual seria o publico alvo desta aplicação, quem tiraria proveito de um serviço deste género e que funcionalidades seriam úteis a esse publico.

De forma a facilitar nesta implementação pensou-se num possível guião de utilização do sistema de forma tentar identificar os requisitos do sistema:

- Utilizador emparelha o seu dispositivo de monitorização ao seu telemóvel e acede ao sistema Life Guardian System.
- Com uma determinada frequência o dispositivo envia um pacote de dados com as medições sensoriais obtidas para o *back-end* (via API).
- O sistema ao receber os dados, armazena-os e, com base nos valores nominais, informa o utilizador do grau de cautela necessário a ter.
- Após um período de tempo o sistema volta novamente a avaliar de forma assíncrona, com base em todos os pacotes recebidos até a data, e no caso de o considerar necessário, envia uma notificação ao utilizador.
- O sistema notifica uma lista de contactos de emergência a pedido do utilizador.

Complementou-se a informação de este guião com um diagrama de casos de utilização:

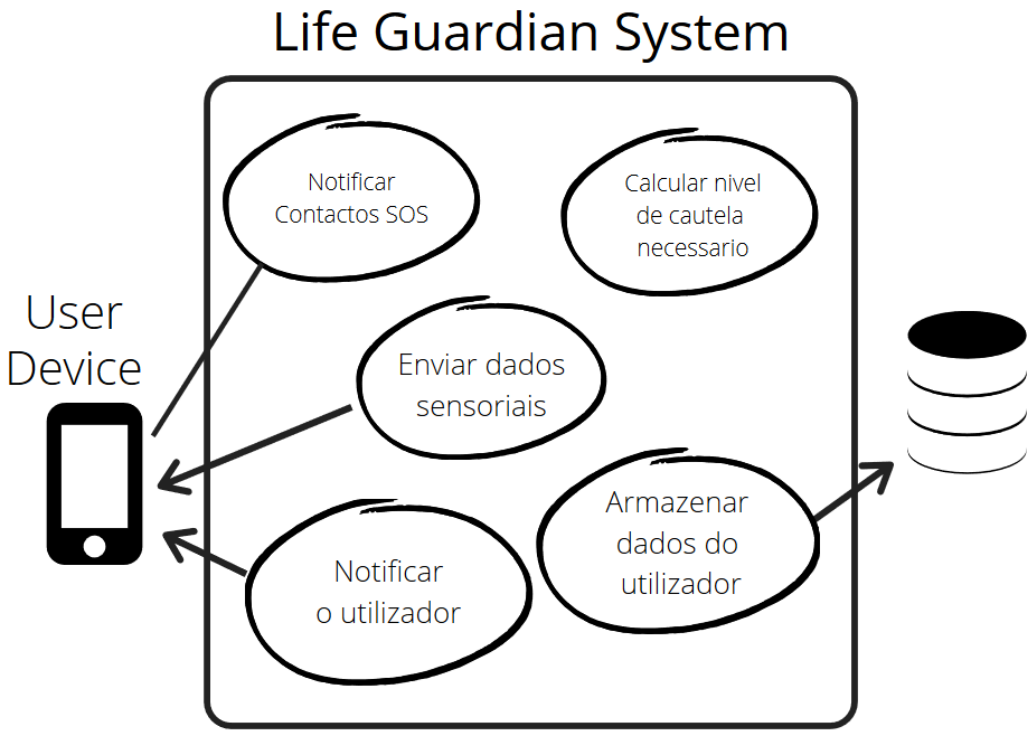

Figura 3.2: Diagrama de casos de utilização

De notar que neste diagrama são omitidas as funcionalidades CRUD que são dadas como uma garantia no sistema.

Após alguma reflexão sobre o guião, o diagrama exposto na figura 3.2 e possíveis eventos que ocorram concluiu-se que este sistema deve ser capaz de:

1. Manter um registo dos seus utilizadores e das suas informações pessoais (incluindo contactos SOS) - Funcionalidades CRUD

2. Garantir a segurança dos dados de cada utilizador - Encriptação de dados e mensagens

3. Receber e armazenar dados biométricos dos utilizadores - Sistema de base de dados

4. Lidar com vários tipos de sensores - Abordagem modular ao problema

5. Enviar Notificações quando necessário. - Definição de comportamentos e sinais considerados para envio de notificações.

6. Contactar os contactos SOS de um utilizador por mensagem automatizada. - Utilização de sistemas externos.

Optou-se por definir primeiro o ciclo de vida da aplicação e desenhar um conjunto de diagramas para modelar o funcionamento do sistema antes de passar a sua implementação.
São resultado deste exercício as figuras 3.3 e 3.4 apresentadas a seguir:

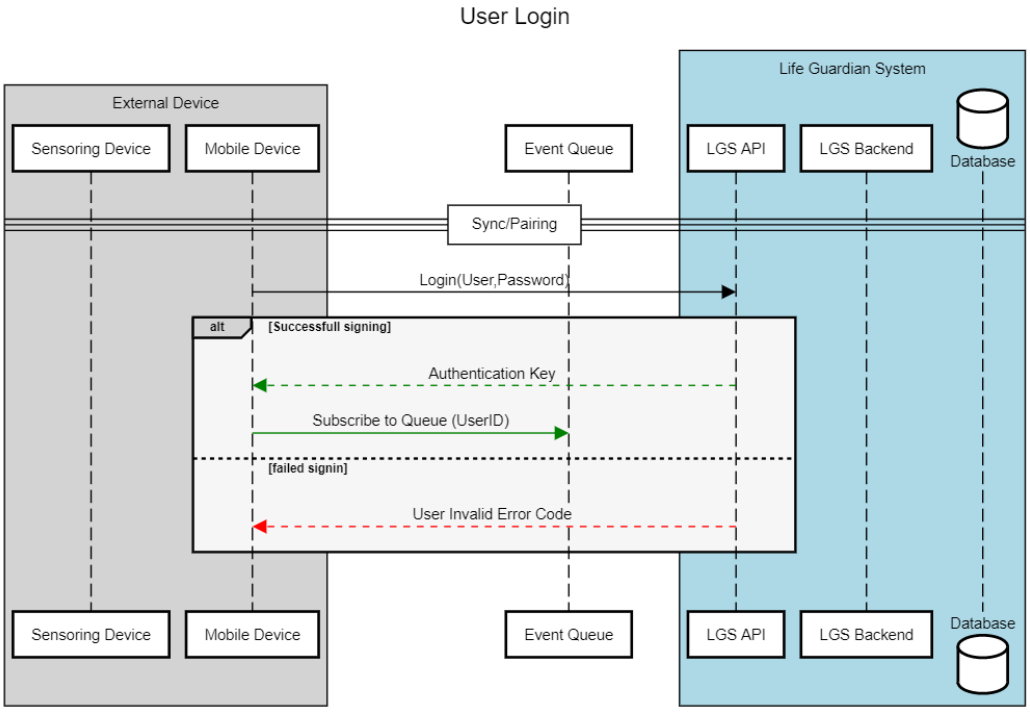

Figura 3.3: Processo de login

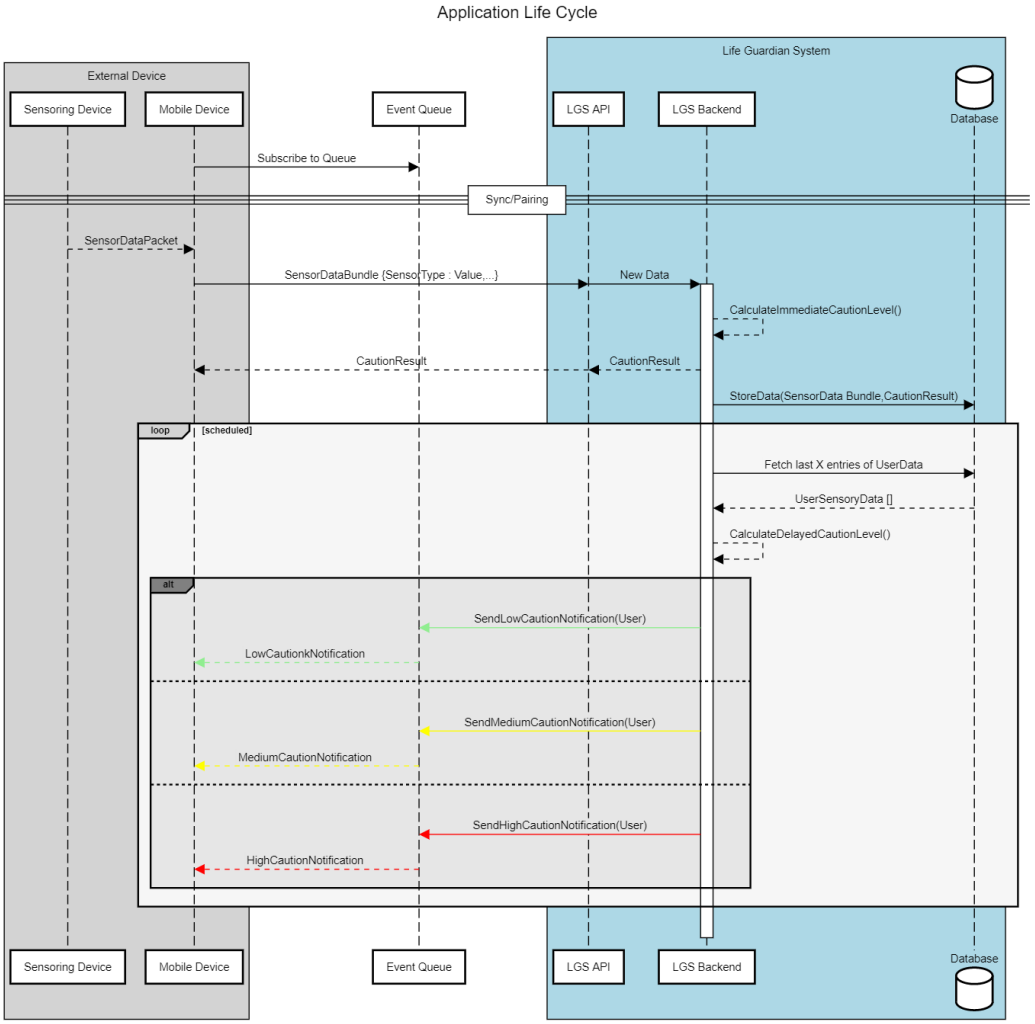

Figura 3.4: Diagrama de sequência do ciclo de vida da aplicação

Estes dois diagramas de sequência tentam transmitir a visão de como será utilizada a aplicação e quais os resultados expectáveis das acções aqui representadas.

3.2 Fundamentos

Já que se pretendia desenvolver uma API, existiu a necessidade de escolher entre os protocolos de comunicação para serviços web SOAP e REST, já que foram estas as tecnologias abordadas ao longo do curso.

SOAP é um protocolo de transferência de mensagens em formato XML para uso em ambientes distribuídos. O padrão SOAP funciona como um tipo de *framework* que permite a interoperabilidade entre diversas plataformas com mensagens personalizadas. Aplicando este padrão em *Web Services*, usa-se o WSDL para descrever a estrutura das mensagens e as ações possíveis em um *endpoint*. Uma das vantagens de SOAP é o uso de um método de transporte "genérico"para enviar o pedido, desde SMTP até mesmo JMS.

O problema desse padrão, é que ele adiciona um *overhead* considerável, tanto por ser em XML quanto por adicionar muitas *tags* de meta-informação. Além disso, a seriação e desseriação das mensagens pode consumir um tempo considerável.

REST é um outro protocolo de comunicação, baseado no protocolo de hipermédia HTTP. Porém não impõe restrições ao formato da mensagem, apenas no comportamento dos componentes envolvidos.
A maior vantagem do protocolo REST é a sua flexibilidade de formato de mensagens do sistema que se adapta ás necessidade do programador. Os formatos mais comuns são JSON, XML e texto puro, mas em teoria qualquer formato pode ser usado. Isso leva-nos a outra vantagem: quase sempre *Web Services* que usam REST são rápidos.

O problema com REST pode surgir justamente por causa das suas vantagens. Como a definição do corpo de dados fica totalmente a cargo do programador, os problemas de interoperabilidade são mais comuns.

Tendo isto em conta, foi decidido utilizar **Spring Boot**, uma *framework* baseada em Java utilizada para a criação de aplicações e serviços web RESTfull

A escolha desta *framework* advém da forte incidência que existe na linguagem de programação Java ao longo da licenciatura, sendo esta a que trás mais confiança e comforto. **Spring boot** além de ser a *framework* Java mais conhecida com um ecossistema e comunidade activa actualmente, automatiza todo um conjunto de processos que de outra forma ocupariam grande parte do tempo de produção e permite desenvolver sistemas com arquitecturas baseadas em micro serviços simplificando o desenvolvimento e melhorando a escalabilidade.

Como sistema de base de dados relacional foi utilizado **PostgreSQL** devido à sua extensibilidade e, como o nome indica, suporta a sintaxe SQL. De acordo com o *blog* [Bhatia, 2020] todos os grandes sistemas de gestão de bases de dados relacionais (**RDBMs**) utilizam uma variante da linguagem pura de **SQL** alterando apenas as funcionalidades disponibilizadas. Sendo **MySQL** a mais popular e mais utilizada hoje em dia e **PostgreSQL** a mais avançada com um crescimento de popularidade ao longo dos últimos anos.

Ao avaliar as duas optou-se por utilizar **Postgres** a titulo de aprendizagem pessoal e, sendo esta *open-source*, abriria possibilidades para futuras melhorias. Apesar de ser compatível com um menor numero de linguagens programáticas quando comparado a **MySQL**, possui mais *plugins* que lhe permite ter mais funcionalidades.
Esta comparação é feita em maior detalhe na pagina [Harris, 2018].

A estas tecnologias juntamos *RabbitMQ*, sendo este um sistema de envio/recepção de mensagens. Serve como intermediário o qual permite reduzir a carga e os tempos de entrega de mensagens do sistema. Este foi escolhido em relação a outros sistemas de mensagens assíncronas como **ActiveMQ** e **ZeroMQ** apenas por se ter tido acesso a mais conteúdos de aprendizagem, uma vez que o tema de "message broker" é ainda recente para nós.

No artigo [Stiller, 2019] é abordado este tema com maior detalhe, comparando-se este tipo de sistemas a outros como *Kafka* e *Azure Event Hub* que disponibilizam mais funcionalidades em troca de maior complexidade de implementação.

Por se lidar com dados pessoais sensíveis, existe a necessidade de proteger a privacidade do utilizador. Para isto foi utilizado JSON Web Token (JWT) que define uma forma compacta e segura de transmitir informação entre os participantes em forma de objetos JSON sendo a sua utilização mais comum em autorização e autenticação(motivo pelo qual este foi utilizado no projecto)

O utilizador faz o pedido de *login* e se as suas credenciais estiverem corretas, o sistema produz e entrega uma JWT ao utilizador. Cada subsequente pedido ao sistema terá de possuir esta JWT para ser autorizado o acesso.Esta tecnologia retira a necessidade de guardar dados de utilização na sessão para autenticar o utilizador. A escolha desta tecnologia em lugar de tecnologias como **Cookies** ou campos escondidos de sessão deve-se ao facto de esta ficar exposta e ser transmitida sem qualquer tipo de segurança acrescida.

Por ultimo, foram utilizados **JavaMail API** e **Twilio** como serviços que permitem enviar *emails* e realizar/receber mensagens ou chamadas telefónicas, respectivamente.
Apesar de ter algumas limitações referentes aos números de telemóvel que é possível enviar mensagens de texto, **Twilio** permite desenvolver de maneira simples e rápida um sistema de notificações por mensagens de texto independentemente da localização do número que se pretende contactar.
Já o **JavaMail API** foi utilizado pois é uma API própria de **Java** que permite o envio de *emails* de forma simples e sem recorrer a nenhuma API adicional.

3.3 Abordagem

Foi utilizado o padrão de desenho MVC que forma uma divisão entre aquilo que deve ser apresentado/entregue ao utilizador do sistema, daquilo que é a lógica de negócio e a implementação dos serviços.
Este padrão é utilizado em aplicações que sejam destinadas a apresentar informação aos utilizadores seja pela forma de uma pagina web, uma aplicação Android, etc...

Optamos pela sua utilização porque este padrão de desenho alem de ser simples de implementar, permite criar aplicações modulares, facilmente escaláveis, e capazes de introduzir novos conceitos sem alteração dos já produzidos. Tendo ainda a vantagem de livrar-nos da preocupação de produzir um *front-end* para o sistema, podendo este ser implementado mais tarde.

A desvantagem deste padrão advém da forma como este é estruturado, de forma a cumprir este padrão deve se implementar um controlador para cada modelo o que pode levar a projectos extensos com um conjunto de classes que só têm uma ou duas funcionalidades.

Como já foi referido anteriormente na secção de 3.2 foi criada uma API do tipo REST, este tipo de API tira proveito dos métodos HTTP para mapear as funcionalidades do sistema como representado na figura 3.5

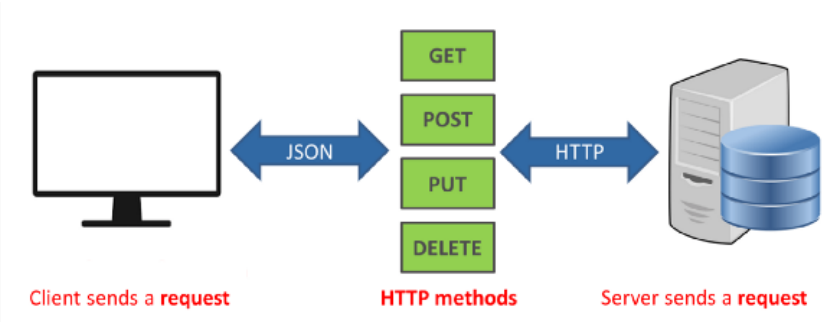

Figura 3.5: API REST

Ditam as boas práticas conforme apresentadas no artigo [Hauer, 2015] que cada método HTTP é destinado a uma funcionalidade especifica.

GET - Método destinado a ler dados

POST - Método destinado a inserir novos dados no sistema

DELETE - Método destinado a remover registos.

PUT - Método destinado a actualizar por completo registos do sistema, caso estes não existam comporta-se como o **POST**

PATCH - Método destinado a actualizar parcialmente um registo

Posto isto, o processo de produção de uma API passa sempre por um conjunto de passos:

1. Levantamento de requisitos (KPI)
2. Desenho do Modelo
3. Implementação

No passo de levantamento dos requisitos é idealizado o projecto e quais são os parâmetros/funcionalidades considerados essenciais. O desenvolver de um projecto sem passar por este processo leva a que as funcionalidades nunca sejam bem definidas tornando muito mais difícil formular um modelo conceptual.

O desenho do modelo ajuda a esquematizar o projecto e clarifica (na medida do possível) se uma determinada implementação é a abordagem mais correcta.

Para o desenho de este modelo foi utilizada a plataforma **Anypoint**. Esta ferramenta de integração é um sistema completo de desenvolvimento, tendo ferramentas tanto para o desenho das API's como para a sua implementação. Utiliza a linguagem RAML que é descendente de linguagens **markup** (**HTML**, **XML**, ...) e permite a definição de API's RESTfull num formato simples e intuitivo.

Apresenta-se a baixo na figura 3.6 o painel da plataforma e como é apresentado o modelo na mesma

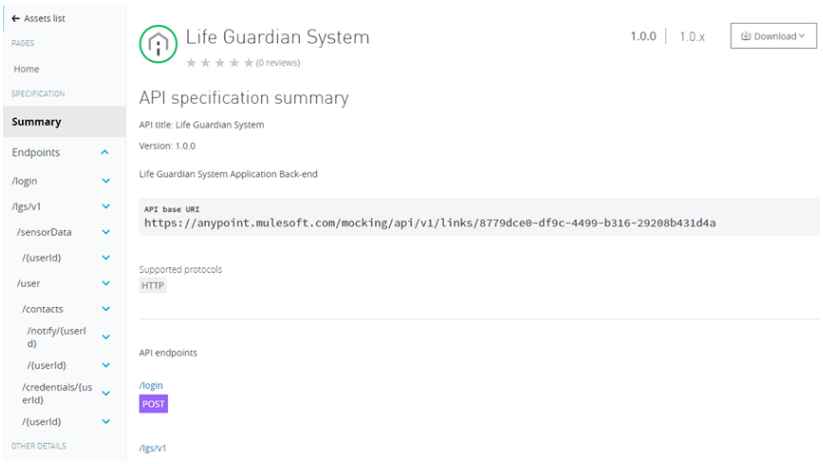

Figura 3.6: Anypoint Platform - API design center

Nota: Utilizou-se este aplicativo apenas para desenhar a API e os seus *endpoints* sendo esta implementada como já foi referido antes em *Java - Spring Boot*. O esquema da API pode ser consultado aqui e neste pode-se visualizar:

- Os vários end-points da API
- Os atributos e a estrutura de mensagens utilizada em cada um desses end-points
- Exemplo de resultados da utilização do end-point
- Exemplos dos possíveis erros que se podem obter quando mal utilizado o end-point

Uma vez concluído este processo de desenho da API passou-se a criação de mecanismos de comunicação e segurança para o sistema.
Aqui, a escolha de JSON Web Token (JWT) passou pela sua forma compacta e fiável de transmitir informação na web no formato **JSON**. Esta *token* é encriptada e contem as informações relativas à sessão, autenticação e *role* de um utilizador.
A figura 3.7 apresenta um *token* antes de ser encriptado e a figura 3.8 apresenta o mesmo *token* após encriptação.

Figura 3.7: Jason Token

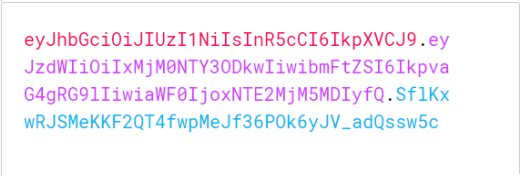

Figura 3.8: Token encriptado

Estes *tokens* tem a particularidade de, ao serem alterados, comprometerem a integridade de informação tornando-se inválidos imediatamente.

Isto provou ser um aspecto acrescido de segurança deveras interessante, contudo, após alguma pesquisa verificou-se que numa versão mais antiga deste sistema existia uma vulnerabilidade relativa a chaves de encriptação assimétricas que tornavam a *token* vulnerável a alterações indetectáveis. Mais sobre este tema é abordado no artigo [McLean, 2020].

Concluindo a idealização de como se iria implementar a estrutura do sistema ficou por elaborar a estrutura para avaliação dos dados e a estrutura que deve ser mantida pelos seus componentes para lidar com cada um dos diferentes tipos de dados biométricos.

Como foi tido em consideração a possibilidade de envio sistemático de notificações ao utilizador por este apresentar valores constantemente fora da escala optou-se por não fazer o lançamento de notificações no imediato, retornando-se apenas o valor obtido para o nível de precaução a ter e uma breve descrição sobre ele e, mais tarde, com base numa amostra de dados recolhidas ao longo de um período de tempo, é então avaliada a necessidade de ser enviada a notificação ao utilizador.

Os modelos de avaliação pretendidos tinham sempre de se basear em fundamentos médicos e, neste caso, por sugestão de um profissional de saúde, apesar do sistema que se pretende implementar não definir directamente risco associado à saúde, foram utilizadas escalas de avaliação de risco como a Escala de Quedas de Morse [NetworkOfCare, 2020] ou a Escala de Branden [IESPE, 2020] que permitem calcular o risco associado a diversos acontecimentos. Estas escalas são orientadas para situações de emergência em que é necessário fazer uma avaliação rápida e concisa sobre o estado de saúde de um individuo tornando as fáceis de implementar informaticamente.

Considerando as escalas mencionadas acima foi possível produzir uma modelo que avalia o grau de cautela existente que o utilizador deve ter em consideração relativamente aos seus dados biométricos que, de forma inicial, incidem apenas sobre a temperatura corporal e/ou a frequência cardíaca do mesmo.

A escala analisa um ou mais factores principais e dependendo do valor que seja fornecido por cada um será pontuada com um valor, sendo directamente proporcional com o grau de cautela a exercer para o determinado parâmetro.

Os valores utilizados são baseados em valores que seriam apresentados normalmente por uma pessoa comum tanto para temperatura ([Kelly, 2006]) como para frequência cardíaca ([MayoClinic, 2018]) e apresentados na seguinte tabela:

	Pontuação			
	0	1	2	5
Temperatura (ºC)	[36-37.9]	[35-35.9], [38-38.9]	[34-34.9], [39-39.9]	<33.9, >40
Frequência cardíaca (bpm)	[60-100]	[50-60[, [101-120[	[40-50[, [120- 150[	<= 40, >=150

Tabela 3.1: Tabela de nível de cautela por parâmetro

Como é possível verificar na tabela 3.1, é atribuída uma pontuação, que varia entre 0,1,2 ou 5, a um ou mais intervalos possíveis para cada um dos parâmetros. Estes valores permitem, caso exista um parâmetro com valor 5, ou seja muito fora do normal, o nível de cautela obtido será influenciado para o de maior nivel já que a forma de obter este nivel, para um utilizador, é através da soma total dos factores analisados.
Isto é, existem dois casos possíveis:

- **É analisada informação sensorial sobre um único sinal vital:**
  Quando este caso se verifica, a escala de medição toma a seguinte forma:

	Nível de cautela			
	Inexistente	Baixo	Médio	Alto
Pontuação total	0	1	2	5

Tabela 3.2: Tabela de medição com um parâmetro

A resposta produzida com a analise de um único parâmetro, apesar de fidedigna, será de menor fiabilidade do que uma resposta analisado os dois parâmetros em conjunto.

- **É analisada informação sensorial sobre ambos sinais vitais:** Quando este caso se verifica, a escala de medição toma a seguinte forma:

	Nível de cautela			
	Inexistente	Baixo	Médio	Alto
Pontuação total	0	[1-3]	[4-6]	[7-10]

Tabela 3.3: Tabela de medição com ambos parâmetros

Analisando a tabela 3.3 verificamos que, os intervalos para cada nível de cautela (baixo, médio ou alto) seguem uma divisão depende do valor máximo que segue a seguinte lógica:

	Nível de cautela		
	Baixo	Médio	Alto
Percentagem sobre o valor máximo (%)	]0-30]	[40-60]	[70-100]

Tabela 3.4: Tabela de percentagens sobre o valor máximo de soma de pontuações

Esta divisão permite dar maior ênfase à necessidade de cautela elevada e além disto, quando existir a introdução de novos parâmetros sensoriais, não existe necessidade de mudar a escala nem de a pontuação atribuída a cada parâmetro, já que se irá adaptar sempre ao valor máximo possível de soma de pontuações.

Já que o modelo descrito acima pode gerar três níveis de cautela diferentes a demonstrar pelo utilizador, foi produzido um algoritmo baseado nestas diferentes escalas.

O algoritmo verifica num espaço temporal, se pelo menos metade dos pacotes de informação sensorial apresentam valores anormais, e no caso de isto se verificar, é enviada uma notificação ao utilizador através do serviço de notificações explicado posteriormente na subsecção 4.1.4.

Devido a maneira como este sistema foi implementado, havia a possibilidade de o utilizador apenas começar a enviar pacotes muito próximo de uma destas tarefas atempadas, resultando num número menor de pacotes que o espectável para a avaliação. Para garantir fiabilidade, foi imposto um ritmo mínimo de dois de pacotes de informação sensorial enviados ao sistema por minuto. Caso esta condição não se verifique, o algoritmo não produz uma notificação.

Este algoritmo é utilizado da mesma forma em três instantes temporais diferentes:

- A cada 30 minutos: Esta chamada ao algoritmo, processa os pacotes enviados pelo utilizador na ultima meia hora para verificação do nível de cautela alto. É verificado se mais de metade do pacotes possuem nível alto, e caso isto se verifique é enviada uma notificação ao utilizador informando-o sobre o mesmo.

- A cada 6 horas: Segue o mesmo pretexto que o ponto anterior, com a diferença deste processar os pacotes das últimas 6 horas e a verificação ser para pacotes de nível cautela médio.

- A cada dia: Este último, a semelhança dos outros dois processa a informação do utilizador enviada num dia completo e verifica se mais de metade dos pacotes enviados são de nível cautela baixo.

Os instantes temporais utilizados e a frequência mínima de envio de informação são baseados em sugestões feitas por profissionais de saúde com extenso conhecimento neste área. Apesar disto, devem ser sujeitos a rigorosos testes de forma a garantir que são adequados pois não existe nenhum tipo de estudo que fundamente a sua exactidão.

Capítulo 4

Implementação do Modelo

Definir um protocolo de mensagens próprio limita a entrada de informação facilitando não só a sua compreensão como também a segurança do sistema. Além disto, era pretendido desenvolver um sistema suficientemente genérico capaz de evoluir sem necessitar de mudanças drásticas na sua estruturação e lógica de processamento. Com isto em mente foram implementados e desenvolvidas as componentes do sistema.

4.1 Arquitetura

Para o desenvolvimento do sistema foi utilizada uma arquitectura baseada em serviços com um padrão de desenho MVC de forma a tentar reduzir a dimensão das classes e manter o principio de responsabilidade singular.
Posto isto, optou-se por dividir o desenvolvimento em quatro categorias de maior importância:

- Base de dados: Classes que definem as funções de acesso à base de dados
- Modelos: Objectos ou estruturas de dados que são utilizadas pelo sistema
- Controladores: Possuem todos os *endpoints* (*URL*) para os serviços separando assim ambos e podendo tornar invisível para o utilizador a lógica de negocio.
- Serviços: Classes que possuem toda a lógica de negocio. São também quem possui acesso ao objecto que define as funcionalidades da base dados

4.1.1 Base de dados

O sistema de base de dados deve armazenar não só os dados dos utilizadores como os dados biométricos adquiridos ao longo do tempo, a estes será adicionado um campo de verificação para que os mesmos dados não sejam processados duas vezes e possivelmente permitir validação para efeitos de aprendizagem automática posteriormente.

O conceito de Utilizador no sistema foi abordado em grande pormenor uma vez que este é o foco principal da aplicação e todos os dados no sistema estão de certa forma interligados a este:

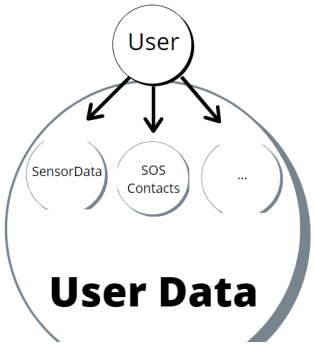

Figura 4.1: Elementos passiveis de pertencer à entidade Utilizador

Foi necessário perceber como organizar estes dados no sistema de base de dados de forma a estes estejam o mais acessíveis possível e que não se receba informação desnecessária em cada pedido efectuado. Posto isto, pensou-se no seguinte:

1. Os dados pessoais do utilizador que sejam constantes (nome, data de nascimento, etc...) devem ser agregados.

2. Os dados biométricos de um utilizador devem ser apenas acompanhados pelo identificador do mesmo, da informação do tipo de sensor e a altura em que foram recolhidos.

3. Os contactos de emergência de cada utilizador por serem diversos e puderem ter vários formatos também devem ser armazenados à parte com um identificador do utilizador, o tipo de contacto e o contacto em si.

Uma vez impostos estes requisitos, modelou-se a estrutura da base de dados:

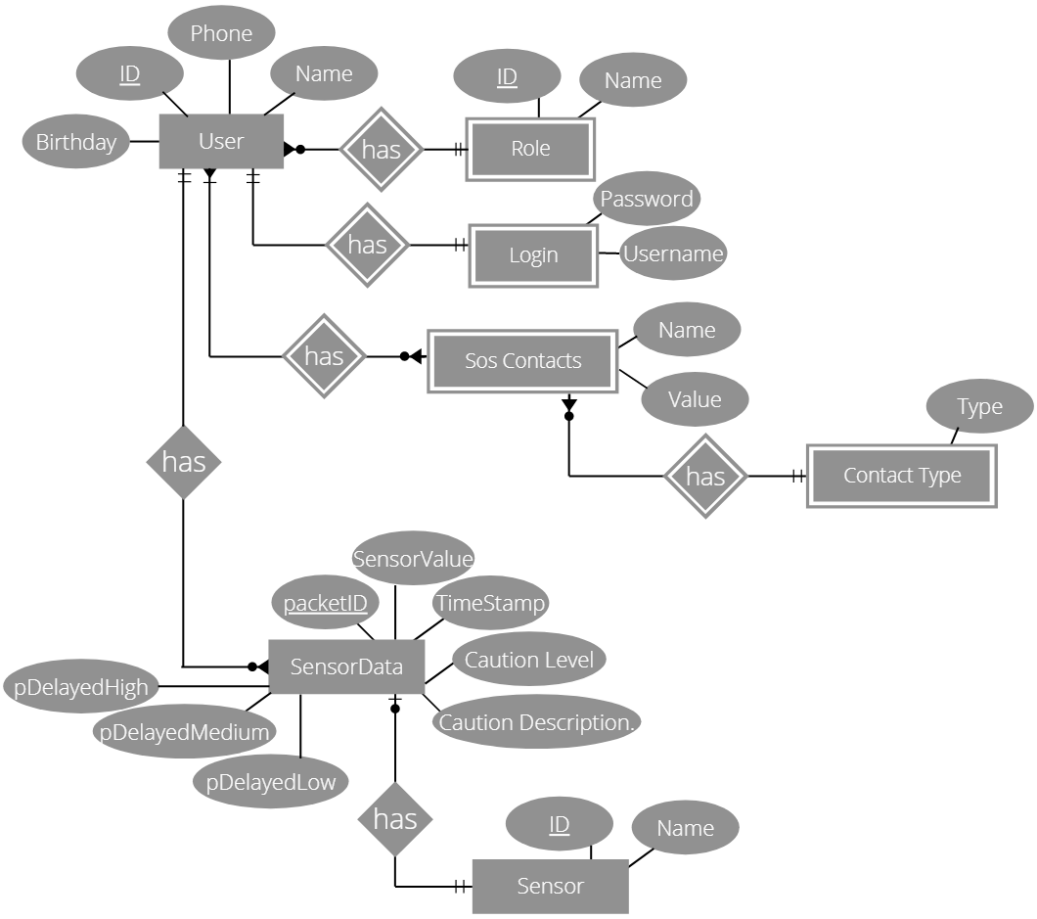

Figura 4.2: Diagrama do Modelo entidade associação

A figura 4.2 representa os conceitos chave do sistema, de notar que neste diagrama optou-se por omitir as entidades que não estão a ser utilizadas de momento de forma a facilitar a sua leitura, estas entidades são apresentadas na figura 4.3.

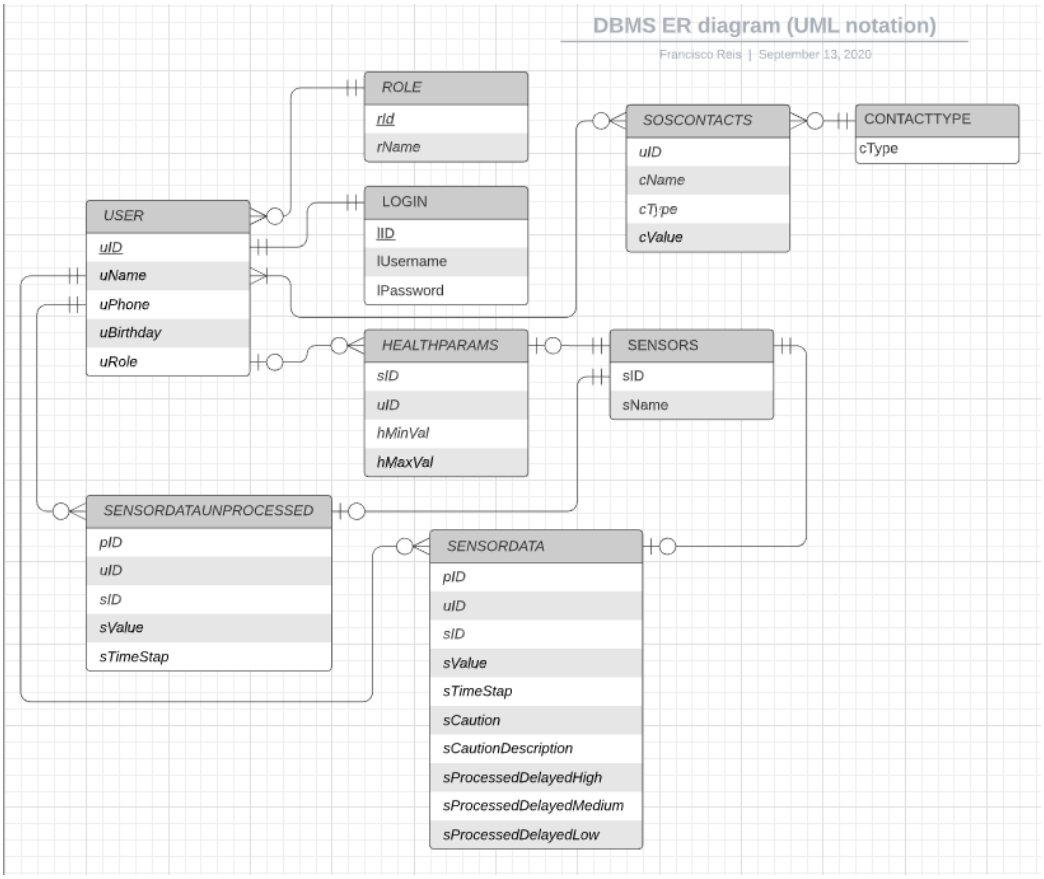

Figura 4.3: UML do modelo entidade-relação da estrutura da base de dados

Optou-se pela separação dos dados de *Login* do utilizador dos seus dados pessoais, desta forma reduzimos o tamanho dos pacotes a enviar ao sistema no momento de acesso, permitindo ainda um pequeno aumento de performance.

De forma a garantir integridade de informação e tipificação de conceitos as tabelas **ROLE** e **SENSORS** contem definidos os cargos e tipos de sensores existentes no sistema respectivamente.

A tabela **HealthParams** destina-se a armazenar informação sobre os valores padrão de cada utilizador, apesar de neste momento não estar a ser utilizada, o seu propósito seria de armazenar um conjunto de valores específicos para cada utilizador que ao longo do tempo se adaptavam as suas especificidades deixando assim de ser genéricos.

A tabela **SensorDataUnprocessed** surgiu de uma funcionalidade que se implementou inicialmente como prova de conceito mas que se acabou por não utilizar, esta tinha como intuito armazenar dados que o utilizador não quisesse que fossem avaliados pelo sistema servindo apenas para visualização num momento posterior.

As tabelas **User** e **SOSContacts** servem para armazenar os dados de um utilizador e dos seus contactos de SOS respectivamente. Esta implementação pensou-se desta forma para que o utilizador não necessite de ter uma lista de contactos SOS na altura de criação de conta, também permite um acesso mais rápido a estes dados com uma *query* a base de dados menos complexa.

A tabela **SensorData** é o local onde serão armazenados todos os dados recebidos relativos a informação biométrica com um indicador sobre o utilizador a que esses dados se referem e uma indicação do momento em que estes entraram no sistema. Os campos *sCaution* e *sCautionDescription* representam as respostas obtidas pelo processamento desta informação sensorial e é armazenada de forma a serem utilizadas pelo processamento baseado num histórico existente. Finalmente, os últimos três campos de esta tabela são valores Booleanos que apenas dizem se um dado pacote já passou por um dos sistemas de processamento assíncrono. Esta abordagem faz com que os mesmos pacotes não sejam processados mais do que uma vez por cada um dos avaliadores.

4.1.2 Modelos

Esta secção pretende explanar as entidades de maior importância para o sistema. Em primeiro lugar foi necessário definir as entidades que seriam alvo de conversão para objectos e começou-se pelas mais óbvias:

- Utilizadores
- Contactos de emergência
- Dados sensoriais

Utilizadores

Os Utilizadores foram mapeados para três tipos possíveis de objecto:

- User
- UserCredentials
- UserView

O primeiro que contêm todos os dados referentes ao utilizador, o segundo contem unicamente as credenciais do mesmo e o último que apenas é utilizado como objecto de visualização que contem informação sobre o utilizador mas sem as credenciais.

Desta forma quando um utilizador pretende criar uma conta é utilizado o modelo que contem toda a informação sobre o utilizador e credenciais inclusive. O segundo modelo que dispõem unicamente as suas credenciais é utilizado quando o utilizador pretende actualizar as mesmas. Por ultimo, o modelo de visualização serve exactamente para este propósito, quando um utilizador pretende obter informação sobre si, por exemplo para a visualização do seu perfil, é utilizado este modelo que não possui as credenciais.
O modelo **User** possui um identificador unívoco do tipo UUID com o propósito de distinguir entre utilizadores.

Contactos de emergência

O conceito de Contacto de emergência optou-se por separar do conceito utilizador pelas razões mencionados no capitulo 3 e apenas fazer referencia a um identificador comum para os associar. Este conceito utiliza dois modelos.

- SOSContactBundle
- SOSContactBundleView

O primeiro contem um *id* que refere ao *id* do utilizador que pretende associar o contacto, um nome próprio referente à pessoa que vai ser contacto de emergência e por ultimo uma lista de contactos ao quais esta pessoa está alcançável, também definida por um modelo próprio e elucidado posteriormente. O segundo modelo deste conceito, possui a mesma informação que o anterior à excepção do *id* do utilizador e é utilizado para pretextos de visualização por parte do cliente.

O modelo da lista de contactos, designado por **SOSContactPacket**, referido previamente, segue uma estrutura tipo dicionário, em que existe um tipo de contacto, por exemplo *email*, telemóvel, entre outros, e associado a este esta o valor que o descreve. O tipo de contacto mencionado encontra-se tabelado na base de dados e só é permitida a utilização destes.

Esta implementação permite a fácil inserção/remoção de contactos SOS não só pelo utilizador como também pelo administrador do sistema.

Dados sensoriais

Os modelos referente aos dados sensoriais seguem a mesma lógica que os modelos referentes a contactos de emergência. Existem os seguintes modelos:

- SensorBundle
- SensorBundleView

O modelo **SensorBundle** contem um *id* que serve para identificação unívoca da conjunto de informação, um *id* do utilizador que envia os dados sensoriais e uma lista de informação sensorial também esta definida por um modelo próprio designado por **SensorPacket**. O modelo **SensorBundle-View** como todos os anteriores modelos que possuem a palavra **View** na sua definição refere a informação visível para o utilizador e contem a mesma informação que o modelo **SensorBundle** removendo deste o *id* para identificação unívoca do pacote e o *id* do utilizador já que estes não se pretendem transmitir abertamente.

O modelo **SensorPacket** segue, da mesma forma que o modelo **SOS-ContactPacket**, uma estrutura tipo dicionário, em que a chave é o nome do sensor que envia informação e um valor que descreve o mesmo. O tipo de sensor também se encontra tabelado na base de dados para garantir que os dados guardados e processados são de facto aceites pelo sistema. Esta estruturação permite que os dados transmitidos pelo utilizador sejam escaláveis, isto é, poderão ser enviados por parte do utilizador qualquer número de entradas de informação sensorial, basta esta ser suportada pelo sistema. Além disto, não limita ao utilizador ter possuir todos os tipos de sensores que o sistema processa.

4.1.3 Controladores

Para a definição dos *endpoints* foi necessário ter em consideração todas as funcionalidades previamente definidas tanto no que diz respeito ao acesso à base de dados como também as monitorizações. Sendo assim, foram definidos três controladores diferentes:

- UserController
- SOSContactController
- SensorDataController

UserController

Este controlador está destinado para todas as funcionalidades referentes sobre a gestão de utilizadores do sistema. Através do URL /lgs/v1/user é possível realizar funcionalidades CRUD como por exemplo criar um novo utilizador, obter um utilizador, apagar um utilizador, actualizar a sua informação, entre outros. Cada uma destas funcionalidades está associada a um dos diferentes métodos HTTP que melhor se adapta.

Segue uma tabela com cada *endpoint* deste controlador, o seu método *HTTP* e a sua utilidade correspondente:

Endpoint	**Metodo HTTP**	**Utilidade**
lgs/v1/user	POST	Inserir um utilizador novo
lgs/v1/user	GET	Obter todos os utilizadores
lgs/v1/user/{id}	GET	Obter informação sobre um utilizador identificado por um id
lgs/v1/user/{id}	DELETE	Apagar um utilizador
lgs/v1/user/{id}	PUT	Actualizar informação dum utilizador identificado por um id
lgs/v1/user/credentials/{id}	GET	Obter as credenciais dum utilizador identificado por um id
lgs/v1/user/credentials/{id}	PUT	Actualizar as credenciais dum utilizador identificado por um id

Tabela 4.1: Tabela *endpoints* **UserController**

SOSContactController

Como o nome indica, o controlador esta destinado a administrar as funcionalidades referentes com os contactos de emergência, tais como, adicionar, obter, apagar ou actualizar um contacto de emergência. Tal como o controlador prévio, cada uma destas funcionalidades está associada a um dos diferentes métodos HTTP que melhor se adapta e é acedido através do URL /lgs/v1/user/contacts.

Alem das funcionalidades mencionadas previamente, este controlador disponibiliza um método **GET** HTTP que permite notificar um contacto de emergência. É possível utilizar esta funcionalidade utilizando o URL /lgs/v1/user/contacts/notify/{id}, em que o *id* recebido como parâmetro no URL se refere ao *id* do utilizador pretende notificar os seus contactos.

Segue uma tabela com cada *endpoint* deste controlador, o seu método *HTTP* e a sua utilidade correspondente:

Endpoint	**Metodo HTTP**	**Utilidade**
lgs/v1/user/contacts	POST	Inserir um novo contacto SOS
lgs/v1/user/contacts	PUT	Actualizar informação dum contacto SOS
lgs/v1/user/contacts	DELETE	Apagar um contacto SOS
lgs/v1/user/contacts/{id }	GET	Obter os contactos SOS dum utilizador identificado por um id
lgs/v1/user/contacts/notify/{id}	GET	Enviar notificação aos contactos SOS dum utilizador identificado por um id

Tabela 4.2: Tabela *endpoints* **SOSContactController**

SensorDataController

Este controlador esta disponível para realizar funcionalidades referentes com o envio, obtenção ou remoção de informação sensorial. É de referir que, aquando o envio de um novo pacote de informação sensorial, é neste momento que o sistema avalia o nivel de cautela imediato e informa ao utilizador do mesmo. Para aceder à funcionalidades do mesmo é necessário utilizar o URL /lgs/v1/sensorData.

Segue uma tabela com cada *endpoint* deste controlador, o seu método *HTTP* e a sua utilidade correspondente:

Endpoint	**Metodo HTTP**	**Utilidade**
lgs/v1/sensorData	POST	Inserir nova informação sensorial
lgs/v1/sensorData/{id}	GET	Obter informação sensorial dum utilizador identificado por um id
lgs/v1/sensorData/{id}	DELETE	Apagar informação sensorial sobre um utilizador identificado por um id

Tabela 4.3: Tabela *endpoints* **SensorDataController**

Nota: É necessário destacar que, apesar da maioria das funcionalidades estarem disponíveis para os utilizadores, algumas funcionalidades mencionadas previamente estão disponíveis unicamente para administradores do sistema. Esta filtragem é realizada através da utilização da anotação **@PreAuthorize** em que recebe como paramento a *role* que pode aceder ao método. A atribuição desta *role* é mencionada na secção 3.3

Login

Existe ainda um outro *endpoint* localizado no URL /login que permite um utilizador realizar a autenticação no sistema. Após um utilizador realizar um pedido a este *endpoint*, o sistema verifica a autenticidade das suas credencias e, caso sejam corretas, envia nos *headers* da resposta HTTP a JWT que autoriza a entrada no sistema e a utilização dos outros controladores e por ultimo o *id* do utilizador de forma a poder realizar qualquer pedido subsequente que necessite do mesmo.

Nota: A descrição completa da API é disponibilizada *aqui*

4.1.4 Serviços

Os serviços, como previamente mencionado, são aquilo que define a lógica de negocio do sistema. São estes que vão ser utilizados pelos controladores quando invocados.
Foram desenvolvidos 6 serviços diferentes, os quais são:

- UserService
- SOSContactService
- SensorDataService
- MonitorService
- NotificationService
- SOSContactNotificationService

Os três primeiros serviços (**UserService**, **SOSContactService** e **SensorDataService**) mencionados foram desenvolvidos em conjunto com os três controladores mencionados na subsecção 4.1.3 e estão destinados a responder aos pedidos referentes as funcionalidades Create,Read,Update,Delete (CRUD) existentes no sistema.

MonitorService

Este serviço esta destinado a responder tanto a pedidos de monitorização instantâneo (subsecção 4.2.1) como também invocar os métodos agendados de monitorização baseada num histórico mencionados na subsecção 4.2.2.

NotificationService

Este serviço tem por objectivo notificar o utilizadores para situações de saude que necessitam de ser avaliadas cautelosamente, obtido pela monitorização baseado num histórico referida na secção 3.3.
Utiliza *RabbitMQ* sob a forma de *publish/subscribe* como forma de notificar utilizadores. Para isto, por cada utilizador, é criada uma *queue* no sistema de **RabbitMQ** que é identificada pelo *id* do utilizador o qual envia os dados. Esta *queue* é criada quando for necessário o envio da notificação por primeira vez ao utilizador necessitar de ser alcançado.

Esta *queue* permite que as notificações a ser enviadas sejam destinadas única e exclusivamente ao utilizador que é necessário notificar. A limitação desta abordagem surge quando existir um elevado número de utilizadores no sistema de forma concorrente. Apesar disto, de acordo com a documentação da própria tecnologia [RabbitMQ, 2013], este número de *queues* é dependente da capacidade da própria maquina mas chega a ser um número bastante elevado na ordem dos 32 milhares. A implementação que permite o envio de uma mensagem para uma *queue* especifica é apresentada no seguinte excerto de código:

```
public void send(String QUEUE_NAME, String message) {
    ConnectionFactory factory = new ConnectionFactory();
    factory.setHost(HOST_NAME);
    try {
        Connection connection = factory.newConnection();
        Channel channel = connection.createChannel();
        channel.queueDeclare(QUEUE_NAME, false, false, false,
            null);
        channel.basicPublish("", QUEUE_NAME, null,
            message.getBytes());
        channel.close();
        connection.close();
    } catch (Exception e) {
        throw new ApiRequestException("Failed to send
            Notification");
        }
    }
}
```

De forma a ser recolhida esta notificação, já que esta tecnologia é baseada na lógica de *publish/subscribe* em que existe um produtor de mensagens que publica numa *queue* e existe um consumidor que se subscreve à mesma de forma a receber estas mensagens, é necessário o cliente estar subscrito à *queue* identificada pelo seu *id*.

Segue um excerto de código que permite esta recolha de informação duma *queue*:

```
public String receive(String QUEUE_NAME) {
        final String[] message = new String[1];

        ConnectionFactory factory = new ConnectionFactory();
        factory.setHost("localhost");
        try {
            Connection connection = factory.newConnection();
            Channel channel = connection.createChannel();
            channel.queueDeclare(QUEUE_NAME, false, false, false,
                null);
            Consumer consumer = new DefaultConsumer(channel) {
                @Override
                public void handleDelivery(String consumerTag,
                                           Envelope envelope,
                                           AMQP.BasicProperties
                                               properties,
                                           byte[] body) {
                    message[0] = new String(body);
                }
            };
            channel.basicConsume(QUEUE_NAME, true, consumer);
        } catch (Exception e) {
            throw new ApiRequestException("Failed to receive
                Notification,queue either empty or doesn't exist");
        }
        return message[0];
    }
```

SOSContactNotificationService

Apesar de ter um nome semelhante ao serviço mencionado previamente, este tem uma finalidade diferente. Esta predisposto a enviar as notificações tanto email como SMS, quando possível, aos contactos de emergência dum utilizador que os pretender contactar. Este serviço invoca tanto a API do **Twilio** como tambem a API do **JavaMail**.

De forma a criar um email basta criar uma nova instância de **SimpleMailMessage**, definir a origem, o destinatário, o assunto e por ultimo o conteúdo do mesmo. Para enviar basta chamar o método *send* da classe **JavaMailSender**.
Bastante simples e directo.

A maneira a enviar um SMS é também bastante simples utilizando **Twilio**. Basta definir um remetente, um destinatário e um conteúdo. O remetente e destinatário é necessário utilizar o objetos do tipo **PhoneNumber** em que o seu único parâmetro é uma *string* que contem um numero de telefone valido junto com o indicativo. O conteúdo basta ser uma *string* com a mensagem a enviar.

De forma a enviar o mensagem é necessário chamar os métodos **creator** e **create**, ambos estáticos, da classe **Message** e **MessageCreator** respectivamene. Os parâmetros do método **creator** são os três objetos referidos previamente. Finalmente, o método **create** não possui parâmetros.

Ambas as formas de notificação são apresentadas no seguinte excerto:

```
SimpleMailMessage mail = new SimpleMailMessage();
                  mail.setTo(sosContactPacket.getContactValue());
                  mail.setFrom("noreplay@lgs.com");
                  mail.setSubject("SOS Alert");
                  mail.setText(SOSNotification.toString());
                  javaMailSender.send(mail);

PhoneNumber to =
               new PhoneNumber(sosContactPacket.getContactValue());
PhoneNumber from =
               new PhoneNumber(twilioConfig.getTrailNumber());
                   String body = SOSNotification.toString();
                   Message.creator(to, from, body).create();
}
```

4.2 Algoritmos

Reserva-se esta secção do relatório para apresentar os principais algoritmos produzidos neste projecto, estes são os algoritmos de avaliação do nível de cautela que um utilizador deve exercer perante a comparação dos dados que o utilizador envia e os dados tomados pelo sistema como padrão.

4.2.1 Nível de cautela imediato

De forma a desenvolver o algoritmo que permite calcular o nível de cautela imediato explanada na secção 3.3 para que seja escalável ao ponto de poderem ser introduzidos novos parâmetros vitais sem necessidade de redesenhar o algoritmo decidimos utilizar o conceito de *Reflection*.

Este conceito em **Java** é utilizado para examinar ou modificar comportamentos de métodos, classes e interfaces em *runtime*. A sua utilização é vantajosa no sentido que permite numa aplicação a utilização de classes definidas pelo programador através da criação de instâncias extensíveis de objetos usando o seu Fully Qualified Name (FQN).

Esta abordagem por sua vez, pode criar problemas de desempenho devido à criação de um objecto novo por pedido.
Uma solução para este problema poderia passar pela criação de uma *pool* de instâncias que só iria criar uma nova quando estritamente necessário.

Segue um excerto de código que demonstra a utilização de *Reflection*:

```
/**
 * This method cycles over all packets in each bundle and finds
 * a suiting handler to process a given
 * sensor value and provide a caution level value of 1,2 or 5
 * according to the given value.
 */
public void calculateInstantCautionLevel() {
    cautionLevel = 0;
    for (SensorPacket packet :
        sensorDataBundle.getSensorData()){
        try {
            Class<?> handler =
                        Class.forName("com.LGS.model.SensorHandlers."
                            + packet.getSensorType());
            cautionLevel +=
                    ((SensorHandler)
                        handler.getConstructor().newInstance()).
                calculateCautionLevelForSensor(packet.getSensorReading());
        }catch (Exception e){
            throw new ApiRequestException("Unsupported sensor");
        }
    }
    this.cautionDescription =
        calculateCautionLevelDescription(cautionLevel);
}
```

Do excerto anterior podemos verificar que por cada valor de informação sensorial presente num pacote de informação enviado pelo utilizador, é criada uma nova instância da classe que lhe diz respeito, utilizado o nome do tipo de sensor recolhido e é calculado o nível de cautela para o mesmo.

Posto isto, existe a necessidade de criação de uma classe especifica para cada tipo de sensor suportado pela escala. Para isto, foi criada primeiro uma classe abstracta da qual todas as classes referidas devem estender, representada no seguinte excerto de código:

```
public abstract class SensorHandler {

    public abstract int calculateCautionLevelForSensor(Float
        sensorValue);

    public boolean isBetween(float x, float lower, float upper) {
        return lower <= x && x <= upper;
    }

}
```

Seguem os excertos de código que permite a definição das classes necessárias para suportar a escala de nível de cautela referida na secção 3.3:

```
public class Temperature extends SensorHandler {
    public Temperature() {}

    @Override
    public int calculateCautionLevelForSensor(Float
        temperatureValue) {
        if (isBetween(temperatureValue, 35f, 35.9f) ||
            isBetween(temperatureValue, 38f, 38.9f)) {
            return 1;
        }else if (isBetween(temperatureValue, 34f, 34.9f) ||
            isBetween(temperatureValue, 39f, 39.9f)) {
            return 2;
        }else if (temperatureValue < 34 || temperatureValue >= 40){
            return 5;
        } else { return 0;}
    }
}
```

```
public class Pulse extends SensorHandler {
    public Pulse() {}

    @Override
    public int calculateCautionLevelForSensor(Float pulseValue) {
        if (isBetween(pulseValue, 50f, 59f) ||
            isBetween(pulseValue, 101f, 119f)) {
            return 1;
        } else if (isBetween(pulseValue, 40f, 49f) ||
            isBetween(pulseValue, 120f, 149f)) {
            return 2;
        } else if (pulseValue < 40 || pulseValue >= 150){
            return 5;
        } else { return 0;}
    }
}
```

Ambas as classes necessitam de um construtor *default* para permitir o uso de *Reflection* e implementadas com base no estudo feito e apresentado no capitulo 3.3.

Os valores utilizados como parâmetros nestas classes poderiam ser recolhidos a partir de uma base de dados facilitando a sua alteração futuramente, contudo, foi tido em consideração a possibilidade de existir situações em que utilizar gamas de valores não seja o mais adequado. Com isto em mente, optou-se por deixar a definição de estes parâmetros em conjunto com a sua utilização dentro de cada *class* que implemente a interface *SensorHandler*

4.2.2 Nível de cautela baseado num histórico

Já que este nível é calculado a instantes temporais específicos, é necessário o agendamento de uma tarefa para o mesmo. A solução para isto foi através da utilização da anotação **@Scheduled** que permite exactamente o pretendido. A esta anotação é fornecida uma *string* como parâmetro que significa o *delay* entre execuções. Este parâmetro utiliza o formato ISO-8601 para durações que segue o padrão **PnDTnHnMn.nS**.

Segue um excerto de código que representa o agendamento da tarefa que é executada a cada 30 minutos:

```
@Scheduled(fixedDelayString = "PT30M")
    public void delayedHighCautionLevelMonitor() {
        String riskMeasurement = "sprocesseddelayedhigh";
        for (UUID userID : usersThatRequestedMonitoring) {
            SensorDataBundleView toBeProcessedHigh =
                sensorDataDao.selectUnprocessedSensorDataByUserId(userID,
                riskMeasurement).
                    orElseThrow(() -> new
                        ApiRequestException("Nothing new to be
                        processed"));
            DelayedCautionCalculator delayedCautionCalculator = new
                DelayedCautionCalculator(notificationServiceImplRbtMQ,
                userID, Caution.HIGH_CAUTION, toBeProcessedHigh);
            delayedCautionCalculator.calculateDelayedCautionLevel();
            sensorDataDao.updateSensorDataByUserId(userID,
                toBeProcessedHigh, riskMeasurement);
        }
    }
```

Este excerto de código permite a recolha de informação sensorial que ainda não passou por este tipo de processamento e envia-a para classe que realiza o processamento. As tarefas que são executadas nos outros dois instantes temporais seguem o mesmo conceito e devido a isto não são apresentados os excertos para os mesmos.

Nota: É de realçar novamente que estes tempos utilizados devem ser sujeitos a rigorosos testes de idoneidade.

O algoritmo que realiza o calculo de nível de cautela necessário recebe a informação sensorial referida previamente e verifica se mais de metade deste informação produziu um nível de cautela especifico (dependente da tarefa que o executa). Foi ainda definida uma frequência mínima de envio informação sensorial que deve ser enviado pelo utilizador de forma a possuir um leque de informação sensorial maior que por sua vez permite maior sustento ao nível obtido. Caso ambas condições se verifiquem, é enviada uma notificação ao utilizador através do sistema produzido com **RabbitMQ**.

Segue um excerto de código que representa o algoritmo mencionado previamente:

```
public void calculateDelayedCautionLevel() {
        int packetsAtHigh = 0;
        for (SensorPacket sensorPacket :
            bundleToProcess.getSensorData()) {
            String cautionDescription =
                sensorPacket.getCautionDescription();
            if
                (cautionDescription.equals(cautionType.getCautionDescription()))
                packetsAtHigh++;
        }

        if ((packetsAtHigh > totalNumberOfPackets / 2) &&
            (totalNumberOfPackets >
            cautionType.getMinimumNumberOfPackets())) {
            notificationServiceImplRbtMQ.send(userId.toString(), new
                RiskNotification(cautionType.getCautionDescription()).toString());

        }
    }
```

É necessário salientar que este algoritmo utiliza os pacotes de informação já processados pelo modelo de nível de cautela imediata ((4.2.1)).

Nota: Como mencionado previamente, os instantes temporais utilizados e a frequência mínima de envio de informação são baseados em sugestões feitas por profissionais de saúde. Apesar disto, devem ser sujeitos a rigorosos testes.

Capítulo 5

Validação e Testes

O modelo proposto para a API definido no capitulo 3 é testado dentro da plataforma *Anypoint* que disponibiliza um *Mocking Service* para testar os vários *endpoints* do sistema, esta permite avaliar (com base numa lista de respostas exemplo) o comportamento expectável do sistema mediante um determinado pedido.
Após implementação dos serviços utilizou-se a aplicação **Postman** para fazer pedidos externos ao sistema de forma a testar a sua capacidade de resposta.

São apresentadas uma série de imagens que representam os testes feitos ao sistema através do **Postman**:

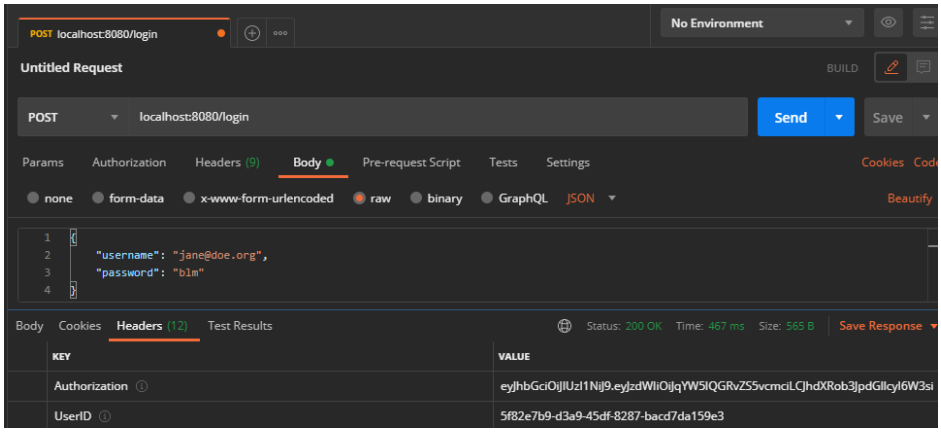

Figura 5.1: Pedido de *login*

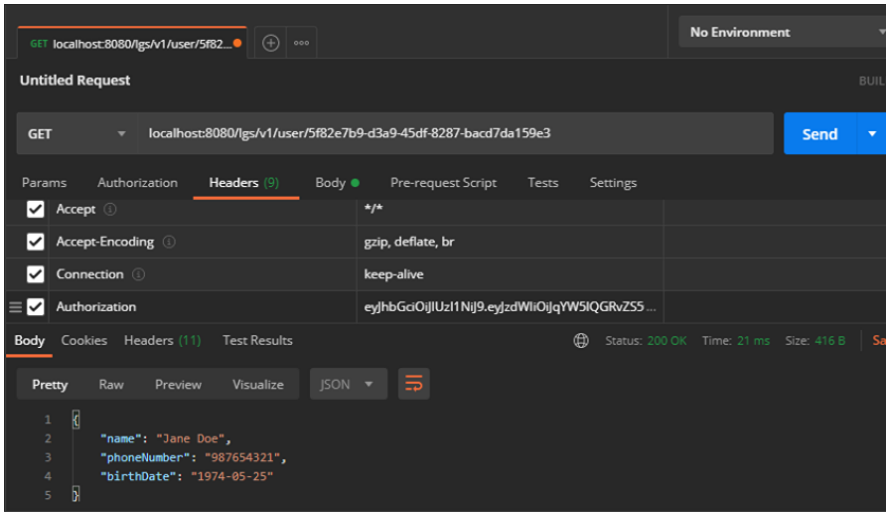

Figura 5.2: Pedido ao controlador do utilizador

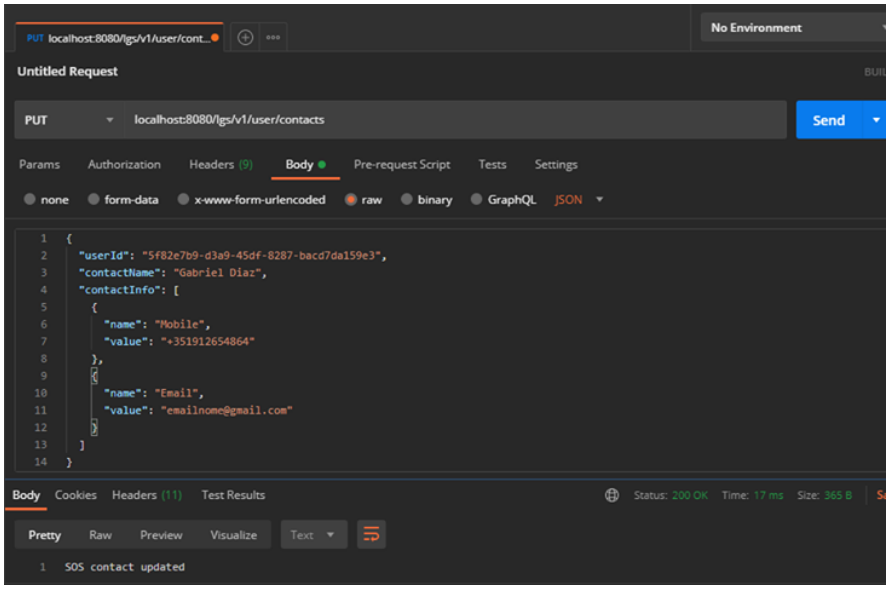

Figura 5.3: Pedido ao controlador dos contactos SOS

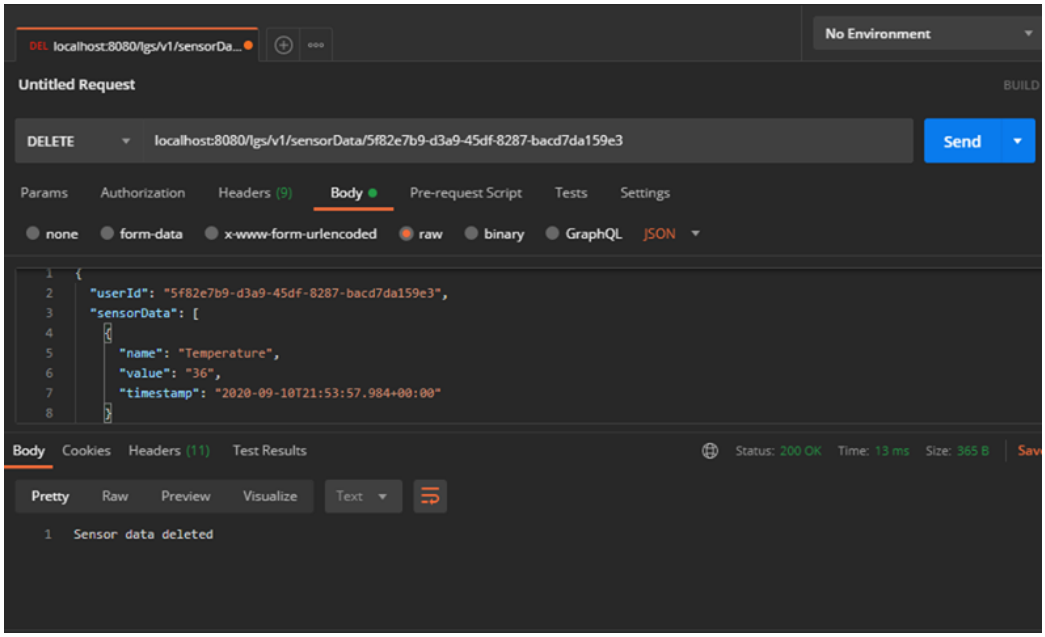

Figura 5.4: Pedido ao controlador dos dados sensoriais

As imagens previamente representam os diferentes tipos de pedidos HTTP possíveis para os diferentes controladores:

- A imagem 5.1 representa um pedido **POST** de *login* ao sistema através do envio de um payload JSON com as credenciais do utilizador. Quando existe *login* bem-sucedido é fornecida uma JWT no *header* ***Authorization*** e o *id* do utilizador no *header* ***UserID***.

- A imagem 5.2 representa um pedido **GET** ao controlador **UserController** da informação do utilizador identificado pelo *id 5f82e7b9-d3a9-45df-8287-bacd7da159e3*. É pretendido destacar a necessidade do *header* ***Authorization*** que inclui a JWT fornecida após *login*.

- A imagem 5.3 representa um pedido **PUT** ao controlador **SOSContactsController** de atualização da informação do contacto SOS do utilizador identificado pelo *id 5f82e7b9-d3a9-45df-8287-bacd7da159e3* e contacto de emergência possui o nome presente no *payload* JSON.

- A imagem 5.4 representa um pedido **DELETE** ao controlador **SensorDataController** de remoção de um pacote de informaçãop sensorial para o utilizador identificado pelo *id 5f82e7b9-d3a9-45df-8287-bacd7da159e3* que possui a informação presente no *payload* JSON.

Nota: Não são apresentadas imagens ilustrativas de todos os métodos para todos os *endpoints* de forma a não tornar este capitulo extenso e repetitivo.

De forma a testar a capacidade de carga do sistema foi utilizada uma ferramenta designada por **JMeter** que serve exactamente para este propósito. Utilizando 100, 1000 e 10000 utilizadores concorrentes, procedeu-se a realizar testes da capacidade de resposta do sistema utilizando a ferramenta mencionada anteriormente:

- 100 Utilizadores concorrentes:

	100 Utilizadores concorrentes					
	Número de pedidos	Latência media (ms)	Latência minima (ms)	Latência máxima (ms)	Desvio padrão (ms)	Tempo de execução (s)
Total	1000	1	1	5	0.69	1

Tabela 5.1: Tabela 100 utilizadores a fazer 1000 pedidos

	100 Utilizadores concorrentes					
	Número de pedidos	Latência media (ms)	Latência minima (ms)	Latência máxima (ms)	Desvio padrão (ms)	Tempo de execução (s)
Total	10000	22	1	381	29.14	3

Tabela 5.2: Tabela 100 utilizadores a fazer 10000 pedidos

	100 Utilizadores concorrentes					
	Número de pedidos	Latência media (ms)	Latência minima (ms)	Latência máxima (ms)	Desvio padrão (ms)	Tempo de execução (s)
Total	100000	36	1	692	47.27	41

Tabela 5.3: Tabela 100 utilizadores a fazer 100000 pedidos

	100 Utilizadores concorrentes					
	Número de pedidos	Latência media (ms)	Latência minima (ms)	Latência máxima (ms)	Desvio padrão (ms)	Tempo de execução (s)
Total	1000000	43	1	1382	59.87	583

Tabela 5.4: Tabela 100 utilizadores a fazer 1000000 pedidos

- 1000 Utilizadores

	1000 Utilizadores concorrentes					
	Número de pedidos	Latência media (ms)	Latência minima (ms)	Latência máxima (ms)	Desvio padrão (ms)	Tempo de execução (s)
Total	1000	4	1	44	5.25	1

Tabela 5.5: Tabela 1000 utilizadores a fazer 1000 pedidos

	1000 Utilizadores concorrentes					
	Número de pedidos	Latência media (ms)	Latência minima (ms)	Latência máxima (ms)	Desvio padrão (ms)	Tempo de execução (s)
Total	10000	205	2	1077	138.65	4

Tabela 5.6: Tabela 1000 utilizadores a fazer 10000 pedidos

	1000 Utilizadores concorrentes					
	Número de pedidos	Latência media (ms)	Latência minima (ms)	Latência máxima (ms)	Desvio padrão (ms)	Tempo de execução (s)
Total	100000	393	2	1745	146.16	42

Tabela 5.7: Tabela 1000 utilizadores a fazer 100000 pedidos

	1000 Utilizadores concorrentes					
	Número de pedidos	Latência media (ms)	Latência minima (ms)	Latência máxima (ms)	Desvio padrão (ms)	Tempo de execução (s)
Total	1000000	493	1	7057	292.40	522

Tabela 5.8: Tabela 1000 utilizadores a fazer 1000000 pedidos

- 10000 Utilizadores

	10000 Utilizadores concorrentes					
	Número de pedidos	Latência media (ms)	Latência minima (ms)	Latência máxima (ms)	Desvio padrão (ms)	Tempo de execução (s)
Total	10000	691	2	3259	818.49	5

Tabela 5.9: Tabela 10000 utilizadores a fazer 10000 pedidos

	10000 Utilizadores concorrentes					
	Número de pedidos	Latência media (ms)	Latência minima (ms)	Latência máxima (ms)	Desvio padrão (ms)	Tempo de execução (s)
Total	100000	3952	2	26692	1720.90	47

Tabela 5.10: Tabela 10000 utilizadores a fazer 100000 pedidos

Os testes foram todos realizados na mesma máquina que executa o sistema e possui as seguintes características:

- Processador Intel i7-8700 @3.20GHz
- 16GB RAM

Esta ferramenta de teste cria uma *Thread* nova por cada utilizador, o que implica que em testes com elevado número de utilizadores concorrentes podem existir valores que não se assemelham à realidade devido às limitações da própria máquina.

Já que também é possível existirem clientes móveis para o sistema, foi construída uma aplicação *Android* de forma a testar as funcionalidades do sistema. Esta aplicação foi criada exclusivamente como ferramenta de teste da API sendo a sua apresentação gráfica bastante rudimentar.

São apresentadas de seguida imagens ilustrativas das funcionalidades testadas pela aplicação *Android* criada:

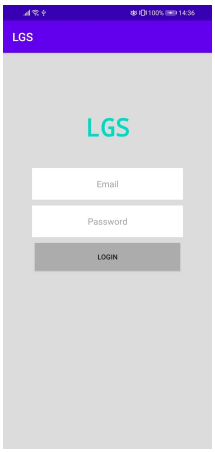

Figura 5.5: Ecrã *login*

Figura 5.6: Ecrã apresentado após *login* bem sucedido

Os ecrãs das figuras 5.5 e 5.6, permitem realizar um pedido de autenticação e após validação correta das credenciais fornecidas, respectivamente.

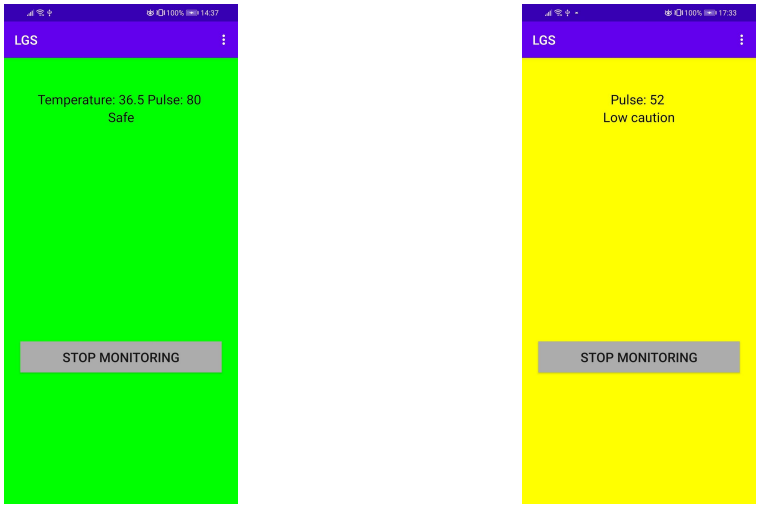

Figura 5.7: Ecrã após pedido de monitorização

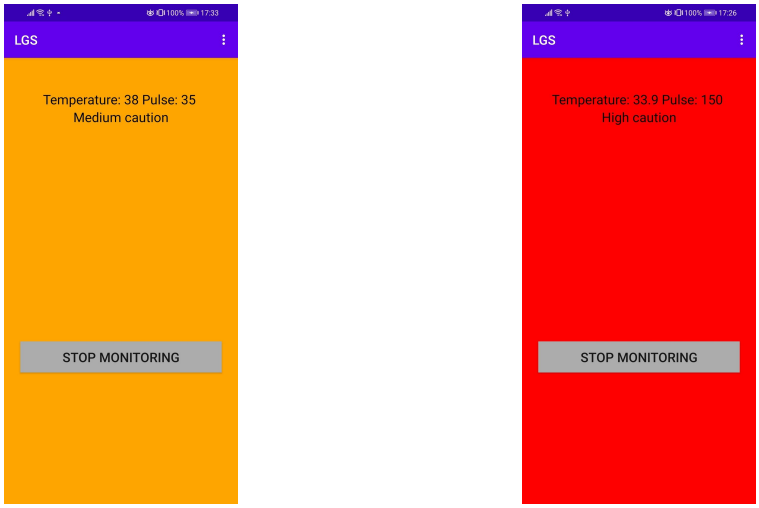

Figura 5.8: Ecrã após pedido de monitorização

Os ecrãs presentes nas figuras 5.7 e 5.8, são apresentados após o utilizador carregar no botão central para realizar um pedido de monitorização e representam os diferente níveis de cautela produzidos da escala imediata.

Esta aplicação envia pacotes com valores aleatórios para os vários tipos de sensores existentes de forma periódica com intervalos de 1 segundo até se pressionado novamente o botão central do ecrã.

Figura 5.9: Perfil do utilizador

Figura 5.10: Histórico do utilizador

As figuras 5.9 e 5.10, representam os ecrãs que apresentam o perfil do utilizador e o seu histórico de dados sensoriais respectivamente.

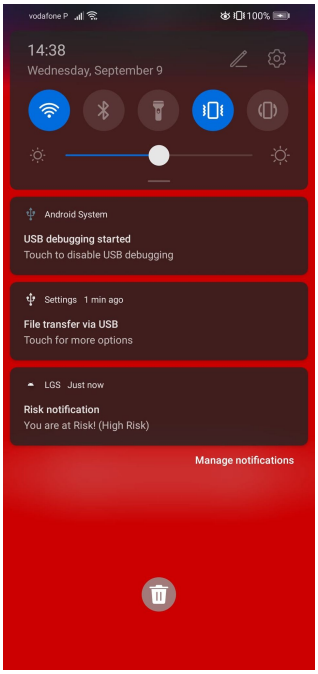

Figura 5.11: Notificação de cautela

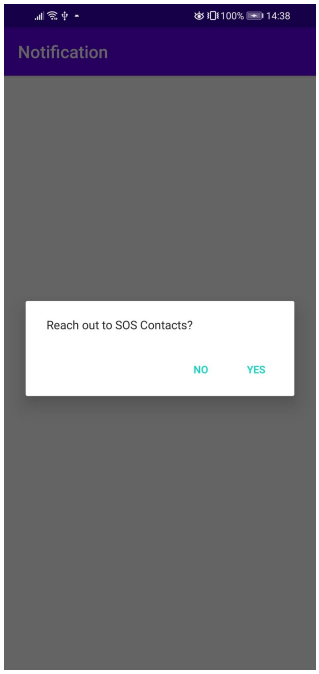

Figura 5.12: Oportunidade de alcance de contactos SOS

A notificação apresentada na figura 5.11 significa que o sistema detectou incongruências no histórico do utilizador e enviou a notificação utilizando o sistema *RabbitMQ* mencionado previamente. Este cliente para receber esta notificação ficou necessariamente subscrito à *queue* que lhe diz respeito.

Por sua vez, a figura 5.12, é apresentada quando o utilizador pressiona na notificação mencionada previamente. É dada a oportunidade ao utilizador de o sistema contactar os contactos de emergência do utilizador.

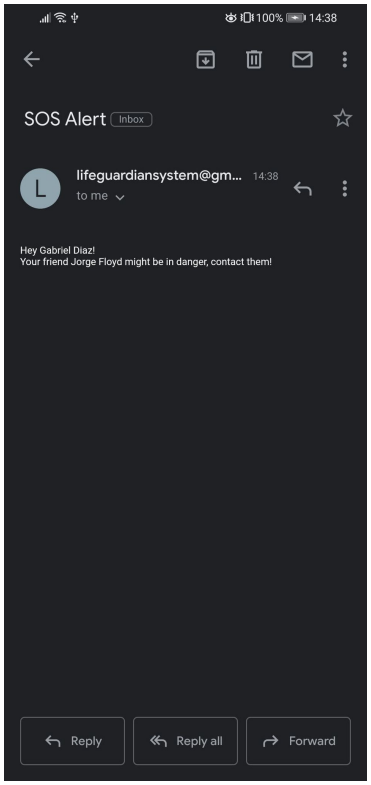

Figura 5.13: Notificação por email

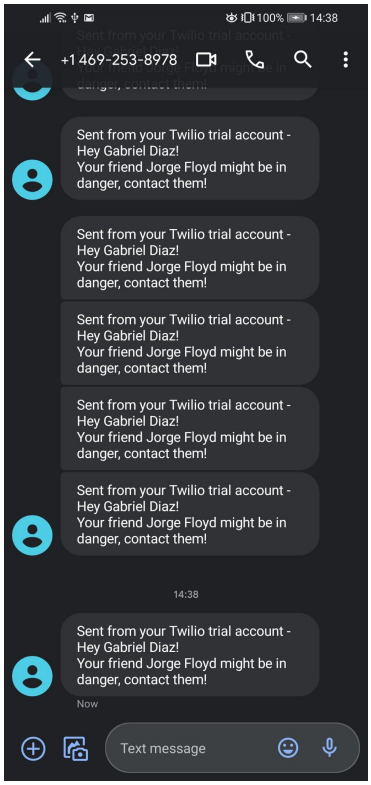

Figura 5.14: Notificação por SMS

Por ultimo, as figuras 5.13 e 5.14, representam as notificações enviadas pela API após o utilizador ter permitido o mesmo. O texto apresentado é personalizado e dependente de uma serie de factores, tais como o nome do contacto e o nome do utilizador.

Nota: É necessário realçar que a maior parte de informação apresentada nas figuras previas resulta de respostas a pedidos HTTP à API desenvolvida.

Capítulo 6

Conclusões e Trabalho Futuro

Este projecto final de licenciatura tinha por objectivo desenvolver um criar uma ferramenta que auxilie os seus utilizadores através da disponibilização de uma API. Para tal, este desenvolvimento apoiou-se numa investigação extensa sobre as melhores abordagens, tanto na área da engenheira como na área médica, para conseguir produzir um produto resiliente e fidedigno.

Com isto em mente, desenvolvemos um projecto que temos orgulho em dizer que é nosso, capaz de evoluir num sistema mais complexo sem a necessidade de reestruturação completa. Pretendemos ainda, apesar de estar concluída esta etapa, continuar a desenvolver este projecto e introduzir novas funcionalidades e melhorias tais como, processamento de maior número de sinais vitais, um sistema de notificações mais robusto, testagem dos algoritmos, entre outros.

O desenvolvimento deste tipo de projectos necessita cautela devido a estar a trabalhar com um tema tão sensível e importante como a saúde pessoal. Considerando isto, é de realçar que este projecto utiliza algoritmos desenvolvidos, com fundamentos e apoio médico, mas que devem ser sujeitos a rigorosa testagem para garantir fiabilidade.

Por ultimo, com o desenvolvimento deste projecto foi possível adquirir e expandir os nossos conhecimento e experiências únicas que permitiram e permitirão crescimento pessoal e profissional.

Bibliografia

[Bhatia, 2020] Bhatia, S. (2020). Postgresql vs mysql: Everything you need to know. https://hackr.io/blog/postgresql-vs-mysql.

[Harris, 2018] Harris, J. (2018). Postgresql vs. mysql: Differences in performance, syntax, and features. https://blog.panoply.io/postgresql-vs.-mysql.

[Hauer, 2015] Hauer, P. (2015). Restful api design. best practices in a nutshell. https://phauer.com/2015/restful-api-design-best-practices/.

[IESPE, 2020] IESPE (2020). Como é a escala de braden e como utilizá-la no ambiente da uti? https://www.iespe.com.br/blog/escala-de-braden/.

[Kelly, 2006] Kelly, G. (2006). Body temperature variability (part 1): A review of the history of body temperature and its variability due to site selection, biological rhythms, fitness, and aging. http://www.altmedrev.com/archive/publications/11/4/278.pdf.

[MayoClinic, 2018] MayoClinic (2018). What's a normal resting heart rate? https://www.mayoclinic.org/healthy-lifestyle/fitness/expert-answers/heart-rate/faq-20057979#:~:text=A%20normal%20resting%20heart%20rate%20for%20adults%20ranges%20from%2060,to%2040%20beats%20per%20minute.

[McLean, 2020] McLean, T. (2020). Critical vulnerabilities in json web token libraries. https://auth0.com/blog/critical-vulnerabilities-in-json-web-token-libraries/.

[MDtechreview, 2019] MDtechreview (2019). Top 10 patient monitoring technology companies - 2019. https://www.mdtechreview.com/vendors/top-patient-monitoring-technology-companies-2019.html.

[NetworkOfCare, 2020] NetworkOfCare (2020). Morse fall scale. https://networkofcare.org/library/Morse%20Fall%20Scale.pdf.

[RabbitMQ, 2013] RabbitMQ (2013). Max messages allowed in a queue in rabbitmq? http://rabbitmq.1065348.n5.nabble.com/Max-messages-allowed-in-a-queue-in-RabbitMQ-td26063.html#a26066.

[Stiller, 2019] Stiller, E. (2019). Rabbitmq vs. kafka. https://medium.com/better-programming/rabbitmq-vs-kafka-1ef22a041793.

[Taleb, 2012a] Taleb, N. N. (2012a). *Antifragile: Things that gain from disorder*. Random House.

[Taleb, 2012b] Taleb, N. N. (2012b). *The black swan: the impact of the highly improbable*. Random House.

[Taleb, 2016] Taleb, N. N. (2016). Incerto. In *AntiFragile*, p. 721–762. Random House, Menlo Park, CA.

[VoCare, 2019] VoCare (2019). Vocare: A revolutionary connected multi-functional diagnostic device. https://patient-monitoring.mdtechreview.com/vendor/vocare-a-revolutionary-connected-multifunctional-diagnostic-device-cid-27-mid-6.html.

[Yokota, 2020] Yokota, F. (2020). Value of information literature analysis: A review of applications in health risk management. https://journals.sagepub.com/doi/abs/10.1177/0272989X04263157.