David António Fonseca Almeida

Implementação em tempo real de um sistema de avaliação automática de leitura de crianças

Dissertação de Mestrado em Engenharia Eletrotécnica e de Computadores

Fevereiro de 2018

Universidade de Coimbra

Departamento de Engenharia Eletrotécnica e de Computadores

Faculdade de Ciências e Tecnologia

Universidade de Coimbra

Dissertação de Mestrado em Engenharia Eletrotécnica e de Computadores

Implementação em tempo real de um sistema de avaliação automática de leitura de crianças

David António Fonseca Almeida

Desenvolvido com a Supervisão de

Professor Doutor Fernando Santos Perdigão

e coorientada por

Jorge Daniel Leonardo Proença

Júri:

Professor Doutor Vítor Manuel Mendes da Silva (Presidente)

Professor Doutor Fernando Manuel dos Santos Perdigão (Vogal)

Professora Doutora Carla Alexandra Calado Lopes (Vogal)

Fevereiro de 2018

Agradecimentos

Em primeiro lugar gostaria de agradecer ao meu orientador de dissertação, Prof. Dr. Fernando Perdigão, pelo apoio prestado e inteira disponibilidade ao longo do desenvolvimento desta dissertação. Agradeço igualmente toda a colaboração e auxílio prestado pelo Jorge Proença. A motivação e exigência colocada, pelos dois, permitiu retirar o máximo proveito da dissertação e adquirir novas competências.

Agradeço aos membros da Gestão da Rede Informática do DEEC (GRI), pela ajuda prestada no processo de atribuição de um certificado SSL ao servidor.

Agradeço também a todos os pais, professores e alunos que disponibilizaram o seu tempo para testar o sistema desenvolvido e responder ao inquérito solicitado.

Um agradecimento muito especial à minha família, pois sem eles não teria sido possível chegar a esta etapa. A motivação, valores incutidos e suporte prestados foram essenciais ao longo de todo o processo.

Por último, a todos os meus amigos, que de uma maneira ou de outra contribuíram para a minha formação pessoal e académica e com os quais partilhei inúmeros momentos de alegria e experiências académicas inesquecíveis, um enorme bem-haja.

Resumo

A avaliação da capacidade de leitura de crianças em idade do 1º ciclo de escolaridade é um processo moroso, normalmente realizado através de processos não tecnológicos, que envolvem o apontamento manual da duração de uma leitura, assim como do número de palavras incorretamente pronunciadas pela criança. Desta forma, a implementação de uma ferramenta que consiga, automaticamente, avaliar a capacidade de leitura de uma criança é algo de grande utilidade, quer para o professor, quer para a criança.

Assim, o trabalho desenvolvido nesta dissertação consiste na criação de uma plataforma online, pensada para ser usada por professores do ensino básico, para rapidamente atribuírem testes de leitura aos seus alunos e analisar os resultados obtidos após a realização desses testes. Durante a realização de um teste de leitura, o sinal de fala produzido pela criança é enviado para um servidor que faz a sua segmentação em palavras e as classifica como correta ou incorretamente pronunciadas. Este resultado é usado para calcular um índice de leitura ponderado, baseado essencialmente no número de palavras corretas e no tempo de leitura.

O sistema desenvolvido permite satisfazer pedidos de vários professores em simultâneo, que podem estar em qualquer lugar onde exista internet, e que é compatível com os browsers mais comuns e com diferentes tipos de dispositivos, sejam eles computadores pessoais, smartphones ou tablets.

O servidor é implementado pela tecnologia Nginx, que suporta uma aplicação em Node.js. O Node.js é uma ferramenta que permite desenvolver aplicações *server-side* em JavaScript. No browser do cliente faz-se uso do Web Audio API, uma biblioteca JavaScript que possibilita fazer várias operações com os sinais de fala provenientes do microfone. Do lado do servidor, estes sinais são analisados por um *addon* do Node.js, desenvolvido em C++.

Palavras Chave: Aplicação web, Web em tempo real, Node.js, Web Audio API, Avaliação de leitura de crianças

Abstract

Assessing 1st grade children's reading aloud capacity is a very time-consuming process, typically performed by the teacher without any kind of technological aid. The teacher measures the reading's duration and manually records the incorrect words. Therefore, developing a tool that automatically evaluates the children's reading ability is of the upmost utility.

Hence, the work of this dissertation aims to create an online platform where teachers can sign up and use the system's tools to quickly assign reading tests to their students and analyze the results after their completion. While performing a reading task, the speech signal is obtained and sent to a server that segments the speech signal into several word regions, while detecting if they were correctly pronounced. Having those word regions, the system computes an overall reading score for that test, based on the number of correct words and the duration of that reading.

The implemented platform can simultaneously satisfy requests of several teachers, originated in any part of the internet and it is cross-browser and compatible with multiple devices, such as personal computer, smartphones or tablets.

The server uses the Nginx technology, who is supporting a Node.js application, a tool to create server-side and network applications. The client's browser uses the Web Audio API, a JavaScript library that enables to perform operations on the speech signals that are being captured by the microphone. In the server-side, these signals are analyzed by a Node.js addon, developed in C++.

Keywords: Web application, real-time web, Node.js, Web Audio API, Evaluation of children's reading ability

Índice

Lista de Figuras iii

Lista de Acrónimos v

Introdução 1

1.1 Motivação 1

1.2 Objetivos 1

1.3 Estrutura da Dissertação 2

Tecnologias Utilizadas 4

2.1 Modelo Cliente-Servidor e Protocolo HTTP 4

2.2 Tecnologias de Servidor 6

2.2.1 Node.js 6

2.2.2 Express 9

2.2.3 SQLite 11

2.2.4 Socket.io 13

2.3 Gravação de Áudio no Cliente 15

Sistema LetsRead 20

3.1 Hardware utilizado e caminhos relativos 20

3.2 Estrutura das diretorias 20

3.3 Funcionamento do sistema LetsRead 22

3.4 Software do servidor web 24

3.4.1 www 24

3.4.2 app.js 24

3.4.3 Routes, controladores e respostas do servidor 25

3.4.3.1 Página de índice 26

3.4.3.2 Registo 28

3.4.3.3 Login 29

3.4.3.4 Prof_index 30

3.4.3.5 Adicionar Alunos 31

3.4.3.6 Lista de Alunos 32

3.4.3.7 Perfil do aluno 33

3.4.3.8 Atribuir Teste 34

3.4.3.9 Teste de Leitura 36

3.4.3.10 Editar dados 41

3.4.3.11 Logout 42

3.4.4 Alterações ao recoder.js 42

3.4.5 Addon de reconhecimento de fala 44

3.4.6 Cálculo do índice de leitura 44

3.4.7 Ngnix como *reverse proxy* 45

Testes ao sistema 47

4.1 Testes de funcionalidade 47

4.2 Testes de segurança 48

4.3 *Feedback* dos utilizadores 49

Conclusão 50

Bibliografia 51

Anexo A 56

Anexo B 60

Lista de Figuras

Figura 2.1 – Overview do modelo cliente-servidor. Retirado de [42]............................. 4
Figura 2.2 – Ícone, no browser Google Chrome, representativo da ligação HTTPS. Retirado de [43]. 6
Figura 2.3 - Evolução da popularidade das tecnologias de programação para o período 2013-2017. [10]. 7
Figura 2.4 – Pedidos ao Node.js sem o módulo Express. Retirado de [15] 9
Figura 2.5 – Pedidos ao Node.js com a utilização do módulo Express. Retirado de [15]. 10
Figura 2.6 – Os diferentes estados de uma ligação via socket. Retirado de [44] 14
Figura 2.7 – Pedido de permissão de acesso ao microfone 15
Figura 2.8 – Exemplo de um audio context. Retirado de [32]....................................... 17
Figura 2.9 – Composição dos ficheiros de formato WAVE. [33] 19
Figura 2.10 – Overview do processo desde captura de áudio até à gravação do ficheiro WAV. 19
Figura 3.1 – Diagrama Entidade-Relação da Base de Dados desenvolvida……………23
Figura 3.2 – Overview do processo de pedido-processamento-resposta implementado pelo Express. Adaptado de [35]... 26
Figura 3.3 – Pedido HTTP GET enviado ao servidor 26
Figura 3.4 – Cabeçalho da resposta HTTP enviada ao cliente 27
Figura 3.5 – Página de índice 27
Figura 3.6 – Página de registo 28
Figura 3.7 – Página de login no sistema.. 29
Figura 3.8 – Página de índice após login na plataforma LetsRead................................ 31
Figura 3.9 – Página para registo dos alunos 31
Figura 3.10 – Página onde é feita a listagem dos alunos inseridos no sistema 33
Figura 3.11 – Página pessoal do aluno 34
Figura 3. 12 – Página de atribuição de um teste de leitura a um aluno. 35
Figura 3.13 – Página de realização de um teste de leitura... 37
Figura 3.14 – Interações entre cliente e servidor, via socket, aquando a realização de uma tarefa de leitura. 38
Figura 3.15 – Página de análise dos resultados de um teste de leitura. 39

Figura 3. 16 - Interações entre cliente e servidor, via socket, aquando a análise de uma tarefa de leitura. 41
Figura 3.17 – Página para edição dos dados do professor. 41
Figura 3.18 – Papel do reverse proxy. Adaptado de [41]. 46

Figura 4.1 – Resultados do teste de segurança 49

Lista de Acrónimos

API – Application Programming Interface

HTTP – HyperText Transfer Protocol

HTTPS - HyperText Transfer Protocol Secure

HTML – HyperText Markup Language

CSS – Cascading Style Sheets

TLS – Transport Layer Security

SSL – Secure Socket Layer

FIFO – First-in, First-out

URL – Uniform Resource Locator

SQL – Strucured Query Language

TCP – Transmission Control Protocol

IIR – Infinite Impulse Response

WAV – Waveform Audio File Format

DNS – Domain Name System

IP – Internet Protocol

ISO – International Organization for Standardization

CSP – Content Security Policy

SUS – System Usability Scale

Capítulo 1

Introdução

1.1 Motivação

Esta dissertação aparece no âmbito do Projeto LetsRead[1], desenvolvido pelo Instituto de Telecomunicações em parceria com o Micosoft Language Development Center da Microsoft Portugal.

No Projeto LetsRead pretende-se desenvolver um módulo tecnológico destinado a crianças portuguesas do 1º ciclo de escolaridade (correspondente, portanto, a uma idade compreendida entre os 6 e os 10 anos) que permita, de forma automática, avaliar a capacidade de leitura das crianças e detetar pronunciações incorretas, repetição de palavras, hesitação na leitura, entre outras disfluências, de modo a que seja possível calcular um índice ponderado do desempenho de leitura da criança. As tecnologias investigadas neste projeto, para além de serem úteis no desenvolvimento na capacidade de leitura das crianças, serão também proveitosas para os professores, tutores e pais, pois terão assim mais uma ferramenta que vem complementar o método de avaliação de leitura tradicional, que é temporalmente muito dispendioso.

1.2 Objetivos

Partindo das motivações apresentadas na secção anterior, o objetivo desta dissertação é desenvolver um sistema disponível online que seja capaz de avaliar em tempo real a leitura de uma criança do 1ºciclo de escolaridade, com base na leitura de um pequeno conjunto de frases e pseudopalavras pré-estabelecido. O sistema deve funcionar através da web, ou seja, acessível em qualquer ponto do mundo. O sistema terá que apresentar os vários textos que a criança terá de ler, gravar as respetivas locuções e enviá-las para um servidor, onde será feito o processamento do sinal de fala. Este processamento consiste em alinhar o sinal de fala com o respetivo texto, de modo a que seja possível extrair várias características que servirão para calcular o índice de leitura da criança. Após

[1] https://www.it.pt/Projects/Index/1938, http://lsi.co.it.pt/spl/projects_letsread.html

o cálculo do índice de leitura, o sistema deve ser capaz de apresentar uma página web onde é possível ao professor, tutor ou pai da criança ver o resultado do processamento do sinal de fala, o índice de leitura calculado e algumas das características que foram extraídas da leitura, como por exemplo, o número de palavras corretas minuto.

Para cumprir os objetivos a que esta dissertação se propõe, terá de ser implementado um servidor web, que controlando todo o sistema deve recebe os pedidos dos clientes e retornar-lhes a devida resposta. Serão ainda necessárias várias tecnologias auxiliares, onde se incluem, entre outras, uma base de dados e um *addon* escrito em C++, desenvolvido no Laboratório de Processamento de Sinal do Departamento de Engenharia Eletrotécnica e de Computadores da Universidade de Coimbra.

1.3 Estrutura da Dissertação

Esta dissertação está organizada em 5 capítulos. O presente Capítulo 1 introduz o enquadramento do trabalho desta dissertação, a motivação que levou ao seu desenvolvimento e os objetivos que esta propõe.

No Capítulo 2 são discutidas e explicadas as tecnologias utilizadas durante o desenvolvimento desta dissertação. É introduzido por uma explanação de alguns conceitos relativos a servidor, cliente e à comunicação entre estes, com particular enfâse no protocolo HTTP. Os restantes pontos do capítulo estão divididos em dois grupos: tecnologias usadas no servidor e tecnologias usadas no cliente. Nas tecnologias usadas no servidor irão ser abordadas as seguintes: Node.js e a sua filosofia de programação *event driven*; Express, as funções de *middleware*, *routing* e os *template engines*; SQLite; Socket.io e web *sockets*. Nas tecnologias usadas no cliente será debatido o Web Audio API e o Recorder.js

O Capítulo 3 terá como assunto principal toda a estrutura que envolve o website implementado. Falar-se-á da estrutura das diretorias e dos ficheiros que são necessários ao funcionamento do website, da estrutura da base de dados (tabelas e as suas relações) criada para esta dissertação e será dada uma explicação em termos funcionais e técnicos sobre cada uma das páginas web presentes no website. Neste capítulo será também abordado o funcionamento de um *addon* desenvolvido no Laboratório de Processamento

de Sinal, que trata do processamento do sinal de fala e onde são usadas técnicas de reconhecimento de fala.

No Capítulo 4 constarão os resultados dos testes de funcionalidade e robustez feitos ao website e o feedback que foi revelado pelos testadores do sistema.

Por último, o Capítulo 5 servirá para expor as conclusões e sugestões para trabalhos futuros que permitam melhorar o sistema.

Capítulo 2

Tecnologias Utilizadas

2.1 Modelo Cliente-Servidor e Protocolo HTTP

O funcionamento da web assenta essencialmente em dois tipos de intervenientes, servidores e clientes, bem como na interação entre ambos. A estas interações dá-se o nome de modelo cliente-servidor. Este modelo baseia-se num método de pedidos e respostas, efetuadas quer pelo cliente, quer pelo servidor.

O cliente é o nome dado ao dispositivo que pede um determinado conteúdo ou função ao servidor [1]. Geralmente este pedido é feito através de um web browser instalado nesse dispositivo, como por exemplo o Google Chrome ou o Mozilla Firefox.

O servidor de web pode ser entendido como a máquina onde estão alojados os ficheiros que compõem as páginas de web (ficheiros HTML, CSS, imagens, etc.) que um cliente pretende ver bem como o software que interpreta os pedidos do cliente e envia a devida resposta [2]. Os pedidos e respostas entre cliente e servidor são possíveis devido ao protocolo HTTP[2] (Hypertext Transfer Protocol).

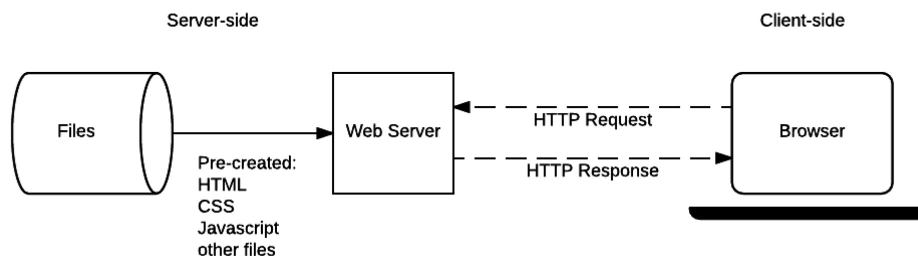

Figura 2.1 – Overview do modelo cliente-servidor. Retirado de [47].

O HTTP é um protocolo *connectionless*, isto é, após o servidor de web ter enviado a resposta a um dado cliente, a ligação entre eles é terminada. Quando o cliente volta a enviar um pedido, a ligação é restabelecida, ou que significa que existe uma nova ligação ao servidor a cada pedido do cliente.

Um pedido feito pelo cliente é denominado por HTTP *request.* Um *request* é constituído pelos seguintes elementos [3]:

[2] https://www.w3.org/Protocols/rfc2616/rfc2616.html

- Um método do HTTP, usualmente um verbo HTTP de onde se destacam dos demais, o GET e o POST e que define a operação que o cliente pretende que o servidor faça. O verbo GET instrui o servidor que o cliente quer receber uma página web, ao passo que o verbo POST avisa o servidor que nesse pedido estão contidos dados introduzidos pelo cliente em determinada página web;
- O *path* da diretoria no servidor onde o recurso que pretende aceder está alojado;
- A versão do protocolo HTTP que está a ser utilizada;
- Outros cabeçalhos opcionais [4];

No caso do método POST, existe ainda um elemento adicional no pedido, onde estão encapsulados os dados a enviar.

A resposta que o servidor envia ao cliente após processar o pedido por este feito é geralmente composta pelos seguintes campos [3]:

- A versão do protocolo HTTP que está a ser utilizada;
- Um código de status, que indica se o r*equest* foi processado ou não com sucesso [5];
- Uma mensagem de status, uma descrição não-autoritativa do código de status que ocorreu;
- Cabeçalhos que fornecem ao web browser do cliente diversas informações [6].
- O próprio recurso que o cliente pretende aceder.

Mais recentemente, as interações ente cliente e servidor não utilizam o protocolo HTTP *per se*, uma vez que a este é juntado o protocolo TLS[3] (Transport Layer System), uma tecnologia que garante que as ligações entre cliente e servidor são seguras e as mensagens trocadas são encriptadas antes de serem enviadas. O uso destes dois protocolos de comunicação em conjunto resulta numa tecnologia nomeada HTTPS[4] (Hypertext Transfer Protocol Secure).

[3] https://tools.ietf.org/html/rfc5246
[4] https://tools.ietf.org/html/rfc2818

Para que um servidor de web possa funcionar com HTTPS, é preciso que uma entidade externa, denominada por Autoridade de Certificação, emita um certificado SSL (Secure Socket Layer) onde são verificadas as origens do servidor de web e que comprova que estas são fidedignas [7]. É através das chaves presentes neste certificado que são gerados os algoritmos que encriptam as mensagens trocadas entre cliente e servidor.

O cliente pode saber se a ligação que estabeleceu ao servidor é feita através de HTTPS num campo existente na barra de pesquisa do seu browser de internet. Caso este apresente o ícone de um cadeado, significa que essa ligação utiliza HTTPS, tal como se pode verificar na seguinte figura:

Figura 2.2 – Ícone, no browser Google Chrome, representativo da ligação HTTPS. Retirado de [48].

2.2 Tecnologias de Servidor

Esta secção serve para introduzir os conceitos sobre os quais se baseiam as tecnologias utilizadas no lado do servidor.

2.2.1 Node.js

No ano de 2008, Ryan Dahl procurava implementar uma nova maneira de programar websites. Quando nesse mesmo ano a Google anunciou o seu navegador da web, Chrome, bem como o Chrome V8 JavaScript *engine* que este utilizava, Ryan Dahl teve a ideia de desenvolver uma biblioteca escrita em linguagem C para implementar *sockets* não-bloqueantes e introduzi-la neste novo JavaScript *engine* [8].

O Node.js[5] vinha assim revolucionar a maneira de fazer aplicações web, e quebrar a tradicional distinção de linguagens de *front-end*, geralmente escritas em HTML e JavaScript, de linguagens de *back-end*, como ASP, .NET ou PHP, uma vez que permitia usar JavaScript tanto no lado do cliente como no servidor. Tais capacidades trouxeram grande popularidade ao Node.js, ao ponto que em Outubro de 2017, o Diretor Executivo da Node.Js Foundation afirmava que todos os dias existem online mais de 8.8 milhões de instâncias do Node.js, número que cresceu em 800000 instâncias nos nove meses anteriores [9]. O Stack Overflow, publicou os resultados de um inquérito a 64000 desenvolvedores, realizado em Janeiro de 2017 e no qual se podia constatar que o Node.js era a *framework* mais utilizada, com 25.8% dos inquiridos a usar o Node.js para desenvolver as suas aplicações [10]. É ainda apresentada em forma de gráfico a evolução da popularidade das tecnologias referidas nos seus inquéritos ao longo dos anos, onde se pode constatar que o Node.js é a tecnologia que mais cresceu nesse período.

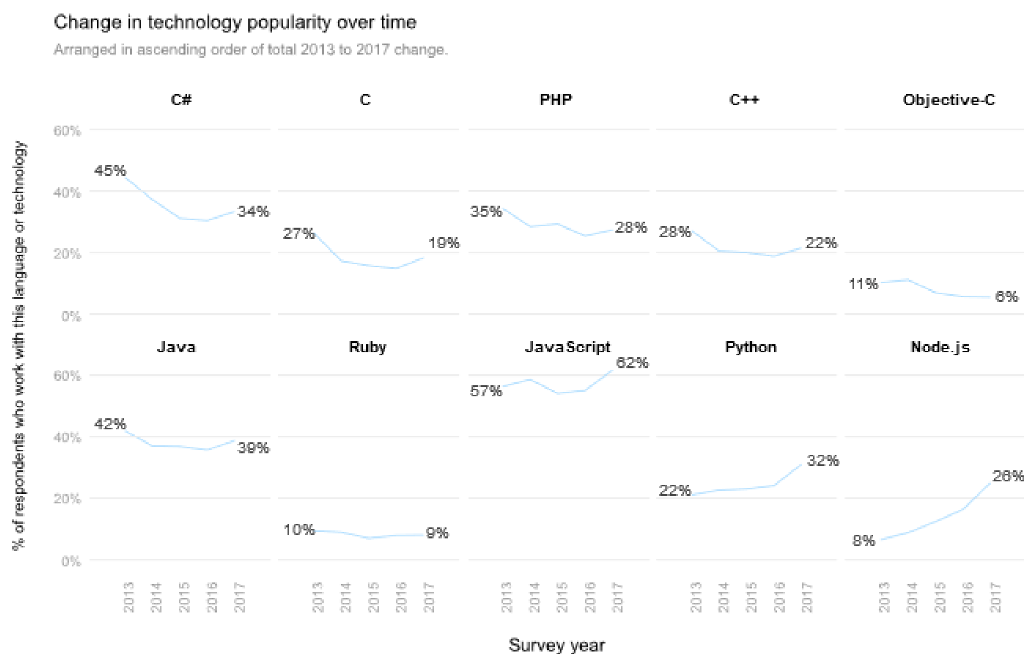

Figura 2.3 - Evolução da popularidade das tecnologias de programação para o período 2013-2017. [10].

O Node.js destaca-se por ser um *runtime* multiplataforma (compatível com Microsoft Windows, Mac OS e Linux) e de código aberto, projetado para desenvolver aplicações *server-side* e de *networking*. Fazendo uso de uma programação *event-driven,*

[5] https://nodejs.org/en/

uma arquitetura denominada por *event loop* e de um modelo de I/O (*Input/Output*) não-bloqueante, o Node.js torna-se então numa ferramenta muito leve e eficiente [11].

A programação *event-driven*, caracteriza-se por ter um fluxo de execução determinado pela ocorrência de eventos. Quando um dado evento é acionado, é executada uma função. Essa função é denominada por *event callback* [12].

A acompanhar a filosofia de programação *event-driven,* aparece o *event loop,* provido pela biblioteca C/C++ libuv. Nos web servers mais tradicionais, cada pedido ao servidor gera um novo *thread*, que consome memória RAM do servidor, o que leva à existência de um limite máximo de pedidos simultâneos. Contrapondo esta abordagem, o Node.js opera apenas em uma *thread*, onde estará a correr o *event loop*. É esta funcionalidade que permite ao Node.js efetuar operações de I/O de modo não-bloqueante, através do descarregamento de operações para o *kernel* do sistema, sempre que possível [13].

Quando o Node.js arranca, é processado o *script* de entrada e iniciado o processamento do *event loop.* Cada fase do *event loop* tem uma fila de espera do tipo FIFO composta pelos *callbacks* a executar. Quando o *event loop* entra numa determinada fase, são feitas as operações específicas dessa fase e de seguida são executados os *callbacks* presentes na fila de espera, até que o último esteja concluído ou que seja atingindo o limite máximo de *callbacks* que podem ser executados. Quando acontece uma destas duas situações, o *event loop* passa para a fase seguinte e assim sucessivamente. [13].

Outra das vantagens do Node.js é que para além das bibliotecas que pertencem ao *core* do Node.js, existe ainda possibilidade em incluir aos projetos outras bibliotecas JavaScript desenvolvidas por outrem. Estas bibliotecas, denominadas por módulos, podem ser encontradas no repositório online Node Package Manager (NPM)[6]. O NPM contém cerca de 475000 módulos escritos em código aberto disponíveis para download, o que o torna o maior repositório de software do mundo [14].

[6] https://www.npmjs.com/

2.2.2 Express

Fazendo uso da potencialidade do Node.js em incluir bibliotecas externas (desenvolvidas por terceiros), utilizou-se o módulo Express, que vem adicionar novas funcionalidades ao Node.js e vem reduzir a complexidade do seu uso, especialmente ao nível do manuseamento de pedidos HTTP efetuados pelo cliente e das consequentes respostas.

As três funcionalidades mais importantes que o Express[7] introduz são o uso de funções de *middleware*, a introdução de *routing* e a capacidade em enviar para o cliente páginas HTML criadas dinamicamente [15].

Quando se cria um servidor web em Node.js sem o uso do Express, todo o código está escrito dentro de uma única função JavaScript, denominada por *request handler function*. Quando um pedido do cliente é recebido, esta função processa o pedido e determina como responder. Este processo pode ser visto graficamente na figura abaixo:

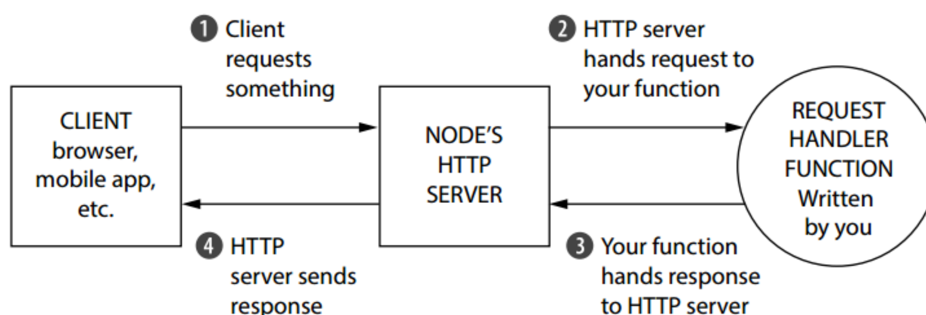

Figura 2.4 – Pedidos ao Node.js sem o módulo Express. Retirado de [15]

Ao usar apenas uma função para tratar todos os pedidos do cliente significa que à medida que vão sendo adicionadas novas funcionalidades ao código do Node.js, o nível de complexidade (e consequente tempo de resposta) desta função irá aumentando na mesma proporção.

Com a utilização do Express, a função *request handler function* é partida numa cadeia de várias funções, denominadas por *middleware,* que vão lidar com uma pequena porção do trabalho que o servidor web tem de executar [15].

---

[7] http://expressjs.com

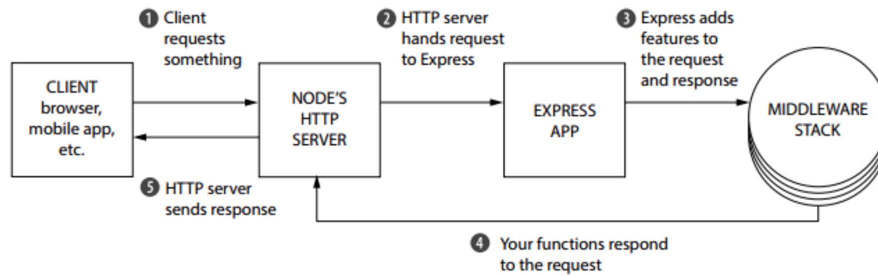

Figura 2.5 – Pedidos ao Node.js com a utilização do módulo Express. Retirado de [15].

As diferenças introduzidas pela utilização do módulo Express ficam bem visíveis fazendo a comparação entre a Figura 2.4 e a Figura 2.5.

Todas as funções de *middleware* presentes nessa cadeia têm como argumento dois objetos JavaScript: o objeto *request* e o objeto *response,* normalmente denominados *req* e *res,* respetivamente.

O objeto *req* contém as propriedades de um pedido à função de *middleware*, como por exemplo, os HTTP *headers*, o corpo da página web, o URL da página web de onde o pedido provém, etc. [16]. O objeto *res* representa a resposta a esse pedido, bem como as propriedades desta, que depois será retornada [16].

Para além desses objetos, as funções de *middleware*, possuem ainda um terceiro argumento, também ele uma função, designado por *next*. Caso numa dada função da cadeia de funções de *middleware* a resposta ainda não tiver sido completada, é evocada a função *next*, que passa os controlos para a próxima função da cadeia e assim sucessivamente, até que esteja pronta a ser retornada [15].

Se acontecer algum erro durante a execução de uma das funções de *middleware,* é automaticamente evocada a função *next*, mas que, em vez de chamar a função seguinte da cadeia, chama uma função de *middleware* especial que recebe o erro e os restantes três argumentos supracitados e, mediante o erro fornecido, determina a resposta que será dada [17].

Como já foi dito, outra das vantagens em se usar este módulo, é a introdução da técnica de *routing*. Esta técnica interliga o pedido do cliente, originário de um dado URL e que contém um verbo HTTP específico, a uma determinada função *route*, que tal como as funções de *middleware,* tem acesso ao objeto *req* e *res* [18].

Desta forma, para todo e qualquer pedido que o cliente faça, existe uma função *route* que o recebe, processa e retorna a resposta ao cliente.

De modo a melhorar a organização da estrutura do código desenvolvido, quando um pedido chega ao *route* apropriado, em vez de ser logo processado nessa função, é redirecionado para um controlador que faz todo o processamento e retorna a devida resposta ao cliente.

Ao nível das respostas aos pedidos dos clientes, o Express traz a vantagem de possibilitar o envio de páginas HTML dinâmicas, através de uma tecnologia apelidada de *template engine*. No âmbito desta dissertação, o *template engine* escolhido foi o pug[8], que para além de ser o *template engine* que o Express instala por omissão, é também aquele com maior documentação disponível [19].

O *template engine* pug, através de uma sintaxe própria, permite que a página HTML que é enviada ao cliente na resposta contenha variáveis que serão alteradas mediante a situação de cada cliente [20], isto é, a página HTML gerada pelo *template engine* é personalizada para cada cliente.

2.2.3 SQLite

Como já referido no capítulo anterior, para além do servidor de web, foi também necessário criar uma base de dados para guardar os dados necessários ao funcionamento do sistema. O gestor de base de dados selecionado no âmbito desta dissertação foi o SQLite[9]. Para além de ser compatível com o Node.js/Express e ter *full support* para estas tecnologias [21], a escolha do SQLite em detrimento de outras opções existentes deveu-se aos seguintes fatores:

- O SQLite é um gestor de base relacional (existem relações entre as tabelas presentes na base de dados) de dados embebido e escrito em código aberto. Ao invés de estar a ser executado num processo *stand-alone*, à parte do servidor web, o SQLite coexiste simbioticamente dentro do processo onde o servidor está

[8] https://pugjs.org
[9] https://www.sqlite.org/

a correr. O seu código está, portanto, embutido no programa (servidor web) onde está a ser utilizado [22].

- O SQLite é um gestor de base de dados muito compacto e leve (apenas necessita de um ficheiro *header* e de uma biblioteca e ocupa menos de meio megabyte no disco do servidor) e configuração-zero, isto é, não necessita de ser instalado antes de ser usado, não é preciso começar/parar/configurar o processo manualmente e não necessita de ficheiros de configuração nem de um administrador que gira as operações sobre as bases de dados [23].
- Este gestor de base de dados foi desenhado a pensar na portabilidade. Pode ser compilado e executado em Windows, Linux, BSD, Mac OS, outros sistemas baseados em Unix como o Solaris ou o HP-UX e ainda em plataformas embebidas como o Symbian ou o QNX. Funciona em ambas as arquiteturas de 32 e 64 bits, quer sejam *big-* ou *little-endian*. A portabilidade do SQLite não se resume apenas às plataformas onde é executado, mas também aos próprios ficheiros das bases de dados, uma vez que são ficheiros binários e, portanto, igualmente compatíveis com todos os sistemas operativos, arquiteturas e ordem de bytes [22].
- O SQLite reconhece praticamente todos os comandos SQL (*Structured Query Language*) existentes para fazer interações com os ficheiros de base de dados, tornando-o assim um gestor de base de dados intuitivo e fácil de usar. A lista completa de comandos SQL que o SQLite permite executar pode ser encontrada no sítio oficial [24].

A integração do SQLite com o Node.js/Express é feita através de um módulo desenvolvido por outrem, denominado por sqlite3[10] e que permite realizar todas as operações sobre o ficheiro da base de dados. Este módulo segue a filosofia do Node.js, uma vez que as operações relativas à base de dados são assíncronas e não bloqueantes, ou seja, o servidor web não fica bloqueado à espera que uma instrução seja completada para avançar para a próxima instrução.

[10] https://www.npmjs.com/package/sqlite3

2.2.4 Socket.io

Como foi referido na Secção 2.1, as mensagens trocadas entre cliente e servidor utilizam o protocolo HTTP, que é *connectionless*. De modo a que o sistema funcione em tempo real, houve a necessidade em introduzir *sockets*, através da implementação do módulo do Node.js denominado Socket.io[11]. Este módulo permite estabelecer uma ligação direta, sincronizada, em tempo real, bidirecional e de baixa latência entre o servidor web.

O Socket.io é um módulo desenvolvido para o Node.js que assenta nos fundamentos do WebSockets API[12], introduzido na especificação do HTML5, e que é uma biblioteca do JavaScript que permite criar *sockets,* mas que, no entanto, não tem compatibilidade com browsers mais antigos. Existe ainda a possibilidade de ter a comunicação bloqueada pela firewall do cliente [25]. Estas contrariedades são ultrapassadas pelo Socket.io por este permitir estabelecer a ligação entre o servidor web e o cliente através de outros mecanismos quando o WebSockets não pode ser usado, como por exemplo, XHR long polling[13] ou o Flash Sockets[14] da Adobe.

Uma ligação estabelecida pelo Socket.io é essencialmente uma conexão através do protocolo TCP (Transmission Control Protocol), isto é, uma ligação ponto a ponto onde os pacotes trocados são entregues ordenadamente e onde existem mecanismos que garantem que não existe perda de informação durante a transmissão [26].

Quando o cliente pretende iniciar uma conexão através de um *socket*, é necessário que um *handshake* seja estabelecido. Em primeiro lugar, o web browser do cliente envia um pedido HTTP especial, onde é requisitado que seja feito um upgrade à ligação HTTP atual, para que esta passe a ser feita através do protocolo TCP. Este pedido HTTP geralmente ocorre como se segue (Retirado de [12]):

```
GET /chat HTTP/1.1

Host: server.example.com

Upgrade: websocket

Connection: Upgrade
```

---

[11] https://socket.io/
[12] https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API
[13] https://en.wikipedia.org/wiki/Push_technology#Long_polling
[14] https://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/flash/net/Socket.html

```
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==

Origin: http://example.com
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 13
```

É de notar que no pedido enviado pelo cliente existe um campo que define uma chave, que é única para cada cliente e que irá servir para confirmar se uma mensagem é enviada para o cliente correto. O servidor web, após receber este pedido, responde ao cliente com a seguinte mensagem (Retirado de [12]):

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
Sec-WebSocket-Protocol: chat
```

A resposta do servidor representa o fim da fase de *handshake* entre as duas partes e o canal de comunicação bilateral é então aberto e é trocado o protocolo pelo qual as mensagens entre cliente e servidor são transmitidas. A partir deste momento, tanto o servidor web como o cliente podem enviar mensagens a qualquer momento, sem ter que esperar um pelo outro. As mensagens trocadas através do *socket* não necessitam de cabeçalhos HTTP nem de *handshakes* adicionais [12]. Quando uma das partes envolvidas fecha o canal de comunicação, a conexão é desligada automaticamente.

O processo de *handshake*, estabelecimento do canal e fim da ligação pode ser visualizado na seguinte figura:

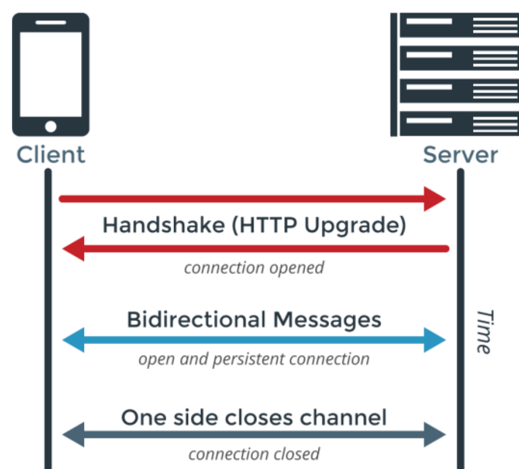

Figura 2.6 – Os diferentes estados de uma ligação via socket. Retirado de [49]

Como já foi referido, estando o canal estabelecido entre servidor web e cliente é possível enviar mensagens entre eles em tempo real. Sendo este um módulo desenvolvido para o Node.js, a transmissão das mensagens é também feita segundo a arquitetura *event-driven*, ou seja, determinada por eventos que ocorram quer do lado do servidor, quer do lado do cliente. O desenvolvedor tem total liberdade para criar eventos para transmissão de mensagens, mas existem alguns eventos que já estão reservados e especificados pelo Socket.io e de onde se destacam, entre outros: o evento *'connect'*, ativado quando um cliente estabelece com sucesso o *socket,* o evento *'disconnect'*, ativado quando o cliente ou o servidor fecham a conexão, o evento *'error'*, ativado pela a existência de algum erro e o evento 'reconnect', ativado quando existe, com sucesso, uma reconexão de um cliente [27].

2.3 Gravação de Áudio no Cliente

Para obter a gravação dos sinais de fala das crianças no navegador da web, foi necessário recorrer a *client-side* APIs (Application Programming Interfaces), isto é, bibliotecas escritas em JavaScript, intrínsecas aos browsers de web e aprovadas pelo World Wide Web Consortium (W3C)[15].

O primeiro passo na gravação de áudio no cliente é requisitar o acesso ao microfone do dispositivo do cliente. Isto é feito através do método *getUserMedia()* da interface MediaDevices, que por sua vez pertence ao Media Capture and Streams API[16]. Este método gera uma notificação no browser do cliente onde é solicitada permissão ao cliente para aceder ao microfone ou câmara do seu dispositivo. Para o browser Mozilla Firefox, a notificação ocorre como representado na seguinte figura:

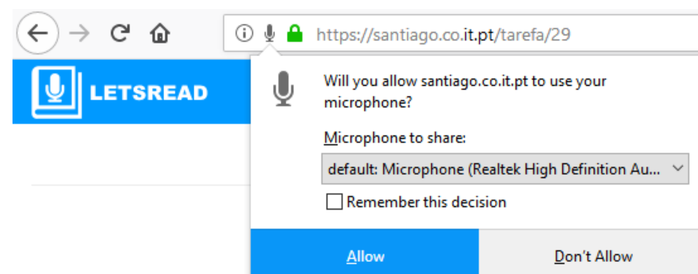

Figura 2.7 – Pedido de permissão de acesso ao microfone

[15] https://www.w3.org/
[16] https://developer.mozilla.org/en-US/docs/Web/API/Media_Streams_API

Caso o cliente conceda permissão para aceder ao microfone ou câmera, o método *getUserMedia()* retorna um objeto do tipo MediaStream, onde constará o conteúdo, ou os *tracks,* do vídeo e/ou áudio que vai ser capturado. Cada *track* pode representar múltiplos canais de entrada, como por exemplo, o canal de áudio esquerdo/direito ou vídeo estereoscópico [28].

No caso de o cliente negar permissão, é devolvida a mensagem de erro consequente e o objeto MediaStream não é criado. É de importância notar que método getUserMedia() requer uma ligação através HTTPS.

Após o acesso ao microfone do cliente ser permitido, o próximo desafio é o de aceder às amostras de áudio, gravá-las e enviá-las posteriormente enviado para o servidor. Numa fase inicial da dissertação, a API usada para aceder às amostras foi o MediaStream Recording API[17]. Este API permite obter os dados adquiridos pelo objeto MediaStream resultante do acesso ao microfone e construía um novo objeto, denominado por MediaRecorder, e sobre o qual era possível fazer processamento e análise ao áudio recolhido e enviá-lo para o servidor web. O funcionamento desta API é baseado em eventos que são ativados pelo cliente.

Quando é ativado o evento *onstart*, os *tracks* começam a ser gravados num Blob [29], um objeto quasi-ficheiro composto por dados em bruto, até que seja ativado o evento *onstop,* que põe fim à gravação de *tracks*. Este evento ativa automaticamente um outro evento, *ondataavailable*, que finaliza o Blob criado que contém as amostras de vídeo e/ou áudio capturadas.

No entanto, este API não oferece suporte para gravação de áudio sem compressão do tipo WAV, essencial para o funcionamento do sistema, nomeadamente ao nível do *addon* usado, pelo que teve que ser trocada pelo Web Audio API[18].

O Web Audio API fornece ferramentas para controlar e processar áudio na web e os seus fundamentos giram em torno do conceito de *audio context*. Um *audio context*

---

[17] https://developer.mozilla.org/en-US/docs/Web/API/MediaStream_Recording_API

[18] https://developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API

engloba todos os nós de áudio que podem ser criados mediante as necessidades do desenvolvedor. Existem quatro tipos de nós de áudio [30]:

- Nós de fonte (*source)*, correspondente às fontes de áudio, que podem ser buffers de áudio, entradas de áudio em tempo real (microfones dos dispositivos) ou provenientes de uma *tag <audio>* do HTML;
- Nós de modificação, onde podem ser alteradas as propriedades das amostras de áudio através da aplicação de filtros, *panners*, etc;
- Nós de análise, onde as propriedades do áudio podem ser lidas;
- Nós de destino, que representam a saída do áudio.

Dentro de um *audio context* os nós de áudio ligam-se entre si, formando um *audio routing graph* [Figura 2.8], que define a maneira como as amostras de áudio circulam desde as fontes de áudio até aos destinos pretendidos, ou seja, desde os nós de entrada até aos nós de destino [31].

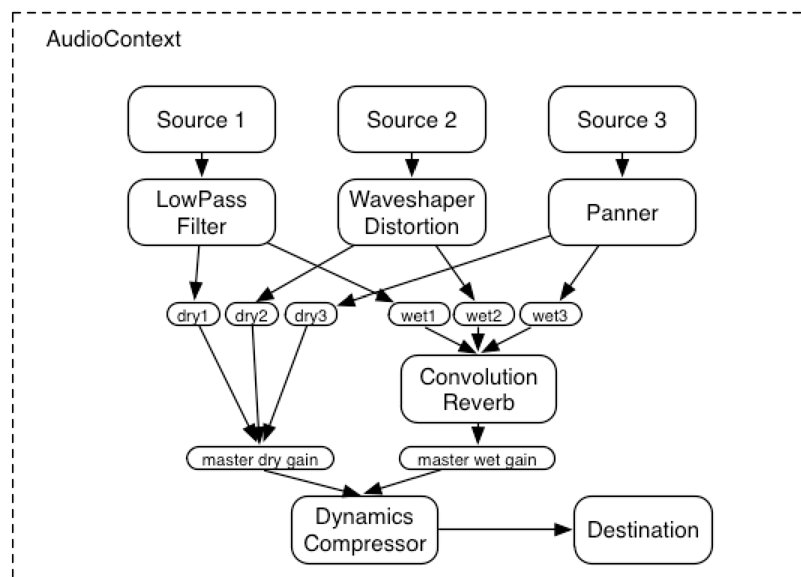

Figura 2.8 – Exemplo de um audio context. Retirado de [32].

O Web Audio API oferece a possibilidade ter vários nós de input e aplicar inúmeros efeitos sobre áudio, permitindo aos desenvolvedores executar operações extremamente complexas no áudio.

É importante referir que o Web Audio API tem suporte nos principais browsers: Google Chrome (versão 14 e adiante), Microsoft Edge, Mozilla Firefox (versão 23 e adiante), Opera (versão 22 e adiante) e Safari (versão 6 e adiante) [31] e que este API

suporta diversos MIME *types* de áudio, como por exemplo, WAV, MP3, AAC e OGG [32].

Em conjunto com o Web Audio API, foi igualmente usado o Recorder.js, um plugin escrito em JavaScript e que facilita a extração das amostras de áudio e reduz em grande parte a complexidade do Web Audio API. O Recorder.js tem acesso ao *audio context* criado pelo Web Audio API e constrói um objeto JavaScript denominado por Recorder, em torno do qual assenta os fundamentos deste plugin.

Quando, dentro do *audio context,* o nó de *source* do Web Audio API é definido, este é passado ao objeto Recorder, onde os seguintes métodos estão implementados:

- *record*, que dá início à gravação do áudio proveniente do nó de *source*. Os *tracks* vindos do microfone vão sendo colocados num buffer que é atualizado de 4096 em 4096 amostras de áudio.
- *stop, onde* a gravação do áudio é finalizada.
- *clear,* que reinicia o buffer e elimina as amostras que este continha.
- *getBuffer,* onde é possível ter acesso às amostras de áudio presentes no buffer e executar operações sobre estas.
- *exportWAV,* este método é particularmente importante, uma vez que é aqui que os dados de áudio em bruto são transformados num ficheiro de áudio do tipo WAV, capaz de ser lido pelos sistemas operativos. Este método vai juntar às amostras contidas no buffer um cabeçalho com 44 bytes e que é composto por variadas informações relativas ao áudio que encapsula (ver Figura 2.9). Os dados ficam no campo "data", com 16 bits por amostra.

Através do uso destes métodos do Recorder.js, o desenvolvedor é agnóstico relativamente às questões relacionadas com as ligações do nó de *source* ao nó de destino do Web Audio API, uma vez que este plugin possui mecanismos para executar essas tarefas.

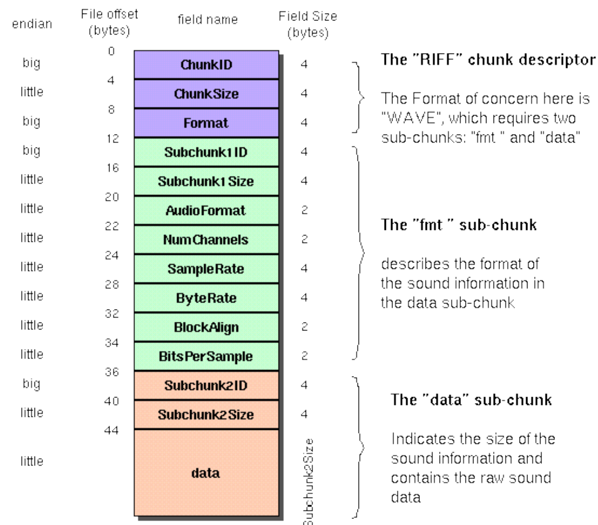

Figura 2.9 – Composição dos ficheiros de formato WAVE. [33]

Para resumir o todo o processo ocorrido desde a captura do áudio pelo microfone até à formação de um ficheiro WAV com esse áudio é pertinente olhar ao seguinte diagrama:

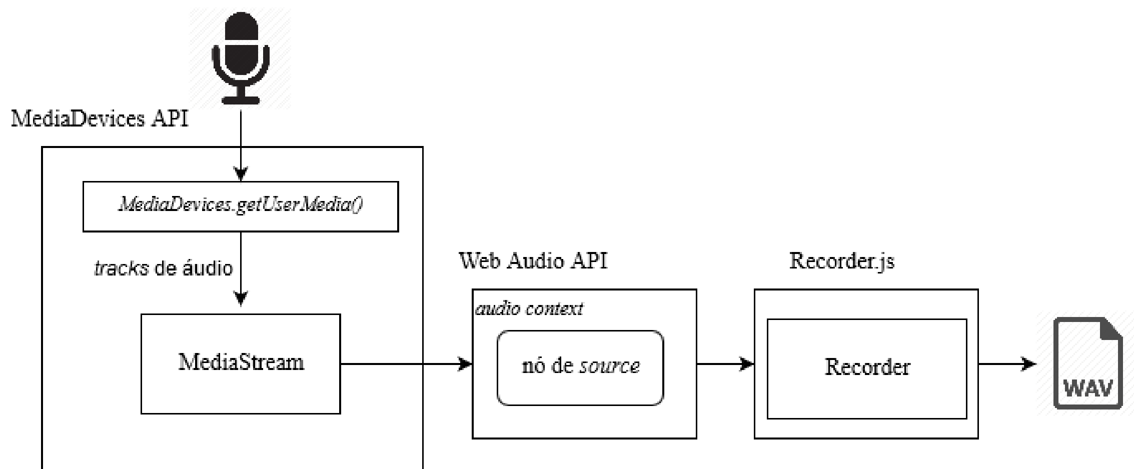

Figura 2.10 – Overview do processo desde captura de áudio até à gravação do ficheiro WAV.

Durante o desenvolvimento desta dissertação foram feitas algumas alterações ao código original do Recorder.js, que serão expostas no Capítulo 3.

Capítulo 3

Sistema LetsRead

3.1 Hardware utilizado e caminhos relativos

Como referido no Capítulo 2, um servidor de web pode ser entendido como o software que determina a resposta aos vários pedidos HTTP/HTTPS dos clientes e como o hardware onde se encontram fisicamente armazenados os ficheiros que constituem esse software, bem como todos os componentes dos websites que este serve.

Começando pelo hardware, durante o trabalho desenvolvido para esta dissertação, fez-se uso de um computador existente no Laboratório de Processamento de Sinal do Departamento de Engenharia Eletrotécnica e de Computadores da Universidade de Coimbra, que possui as seguintes características:

- Processador: Intel Core i5 CPU 760 @ 2.80GHz x 4
- Memória RAM: 3.8 GB
- Disco: HDD 980.1 GB
- Sistema Operativo: Ubuntu 16.04

Este computador tem o endereço de IP 193.136.94.93, que se traduz nas tabelas de DNS (Domain Name System) no domínio santiago.co.it.pt.

É através deste domínio que o website implementado no âmbito desta dissertação está disponível para acesso na web, através do URL https://santiago.co.it.pt/. Além desse URL de *root* (/), foi necessário criar os seguintes caminhos relativos para as diferentes funcionalidades que o website permite fazer: /login, /registo, /contactos, /prof_index, /adiciona, /atribui, /alunos, /alunos/:id, /tarefa/:id, /editardados e /logout, que serão explicados numa secção mais à frente.

3.2 Estrutura das diretorias

Como referido no Capítulo 2, o Node.js é uma plataforma que oferece enorme flexibilidade e oferece ao desenvolvedor total liberdade relativamente à organização dos

diferentes ficheiros que constituem o software do servidor web. O código desenvolvido foi então organizado da seguinte forma:

Nome	Data de modificaç...	Tipo	Tamanho
addon_decoder	21/12/2017 13:38	Pasta de ficheiros	
audios	21/12/2017 13:29	Pasta de ficheiros	
controllers	21/12/2017 13:29	Pasta de ficheiros	
lixo	10/02/2018 21:47	Pasta de ficheiros	
node_modules	21/12/2017 13:37	Pasta de ficheiros	
public	21/12/2017 13:31	Pasta de ficheiros	
routes	21/12/2017 13:31	Pasta de ficheiros	
sockets	21/12/2017 13:31	Pasta de ficheiros	
views	21/12/2017 13:31	Pasta de ficheiros	
app.js	20/12/2017 16:18	Ficheiro JS	4 KB
bd_v4.db	20/12/2017 16:50	Ficheiro DB	72 KB
package.json	20/12/2017 01:41	Ficheiro JSON	1 KB
readingscore_v1.js	20/12/2017 16:44	Ficheiro JS	2 KB
sessions	20/12/2017 18:16	Ficheiro	12 KB
www	11/12/2017 23:09	Ficheiro	3 KB

Figura 3.1 – Estrutura das diretorias

Como é possível observar pela figura anterior, na pasta de *root* encontram-se os ficheiros www, bd_v4.db, app.js, package.json, sessions e readingscore_v1.js. O ficheiro bd_v4.db é onde se encontram fisicamente guardadas as tabelas e os dados que constituem a base de dados criada no âmbito desta dissertação. O package.json é um ficheiro integrante da *framework* Express e que contém uma listagem de todos os módulos instalados, assim como da versão que cada um usa [34]. Um exemplo de uma entrada neste ficheiro pode ser visto de seguida:

```
{
  "name": "Letsread",
  "version": "0.0.0",
  "private": true,
  "scripts": {
    "start": "node www",
  },
  "dependencies": {
     "express": "~4.15.2",
     "express-session": "^1.15.4",
    "express-socket.io-session": "^1.3.2",
    "express-validator": "^3.2.0",
    "pug": "~2.0.0-beta11",
```

```
    "serve-favicon": "~2.4.2",
    "socket.io": "^2.0.3",
    "sqlite3": "^3.1.13"
},
  "devDependencies": {
    "nodemon": "^1.11.0"
  }
}
```

Os restantes ficheiros mencionados serão descritos numa secção mais adiante.

A pasta de *root* subdivide-se noutras 8 diretorias: addon_decoder, onde é armazenado todo o código referente ao *addon* de reconhecimento de fala; áudios, onde serão guardados todos os ficheiros .wav obtidos; a pasta controllers contém os controladores necessários ao funcionamento do sistema; node_modules, onde se encontram instalados todos os módulos usados pelo Node.js; na diretoria public encontram-se os ficheiros estáticos (ficheiros CSS, imagens, etc.) usados pelas diferentes páginas do website; a pasta routes contém o ficheiro onde estão definidas o *routing* da *framework* Express; sockets, onde está disponível o código que envolve o uso de *sockets*; a diretoria views contém os vários ficheiros usados pelo *template engine* pug para gerar páginas HTML dinâmicas.

3.3 Funcionamento do sistema LetsRead

Como já referido no Capítulo 1, foi necessário gerar uma base de dados onde estarão armazenados todos os dados necessários para o funcionamento do sistema. As tabelas criadas e as relações que são estabelecidas entre elas podem ser entendidas de olhando para o esquema relacional [Figura 3.2].

No Anexo A está exposta uma explicação mais detalhada sobre cada uma destas tabelas.

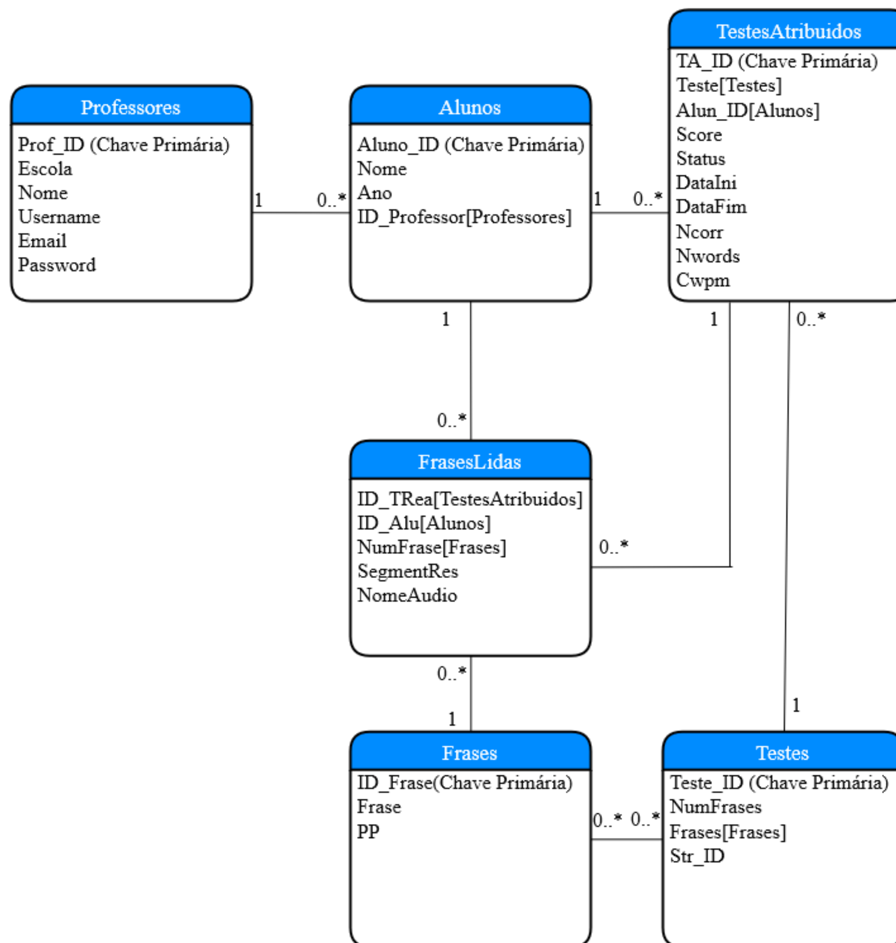

Figura 3.2 – Diagrama Entidade-Relação da Base de Dados desenvolvida.

A necessidade em implementar a base de dados com esta estrutura específica será explicada recorrendo a um exercício de suposição.

Suponhamos que um professor se regista no sistema. A tabela Professores armazena os vários campos com as informações desse professor. Se esse professor tiver 5 alunos aos quais quer fazer avaliação de leitura, terá de os especificar. A tabela Alunos serve para armazenar essa informação e associar o identificador único desse professor aos seus alunos. Depois disso, o professor tem de atribuir pelo menos um teste de leitura: a informação é guardada na tabela TestesAtribuidos, que tem relação com as tabelas Testes e Alunos, isto é, a cada teste atribuído é associado o identificador do teste e o identificador do aluno ao qual foi atribuído. A tabela Testes contém o conjunto de frases e/ou pseudopalavras que compõem um dado teste de leitura (pré-estabelecido) e relaciona-se com a tabela de Frases, onde estão guardadas cada uma das frases e pseudopalvras usadas para avaliação de leitura.

Só depois da atribuição de um teste de leitura a um aluno é que este pode ser iniciado. Pode supor-se que o teste é feito no PC do professor e que o este chama o aluno para um local reservado para realizar o teste. A partir deste momento é o aluno quem controla o avanço do teste. À medida que o aluno vai lendo as frases e/ou pseudopalavras, o sinal de áudio correspondente é enviado e armazenado no servidor, é calculado o resultado do *addon* e é escrita uma nova entrada na tabela FrasesLidas, associada ao identificador do aluno e ao identificador da frase que leu. Após concluir o teste, chega ao fim o envolvimento do aluno. O professor tem de voltar a fazer autenticação no sistema e pode agora consultar os resultados do teste de leitura.

3.4 Software do servidor web

Esta secção serve para explicar o papel que cada um dos ficheiros desempenha dentro do processo do servidor web.

3.4.1 www

Este ficheiro é o ponto de entrada do servidor. Quando o processo do Node.js é iniciado, é evocado este ficheiro, que começa por importar o objeto da aplicação Express. Seguidamente, através das funções de core do Node.js, cria o servidor HTTP, define o porto onde o servidor escuta os pedidos e anexa os *sockets* usados por este servidor HTTP. No caso deste processo de criar o servidor HTTP, definir o porto e anexar o *socket*, este ficheiro contém funções para tratar desses erros e abortar o processo de forma segura.

Só após este ficheiro ser executado é que todos os outros ficheiros que constituem o software do servidor são executados.

3.4.2 app.js

É neste ficheiro que é importada e configurada a *framework* Express. São igualmente importados todos os módulos, previamente instalados via npm, e as restantes bibliotecas necessárias ao seu funcionamento, enumerados de seguida:

- path, um módulo pertencente às funções de *core* do Node.js, que permite manipular os caminhos das diretorias;
- cookie-parser, um *middleware* do Express para manuseamento de *cookies* no cliente;

- body-parser, um módulo muito importante para o funcionamento do sistema desenvolvido, pois adiciona o objeto *body* ao objeto *request* do Express e que facilita o acesso aos dados enviados pelos clientes através dos pedidos HTTP POST;
- sqlite3, o módulo que permite fazer a integração da base de dados criada para este projeto no servidor; express-session e connect-sqlite, módulos para tratar das sessões HTTP dos clientes.
- helmet, um módulo que altera alguns campos do cabeçalho dos pedidos HTTP de modo a que a aplicação seja mais segura.

É também importado o módulo onde estão definidas as funções de *route* implementadas.

De seguida, é criado o objeto da aplicação Express, sobre o qual assenta o funcionamento desta *framework*. Após a criação da aplicação, são-lhe atribuídos vários parâmetros, tais como a definição do *template engine* que está a ser usado, o caminho para o favicon e o *middleware* que processa as sessões HTTP dos clientes e que será discutido com maior detalhe numa secção mais à frente. É ainda atribuído à aplicação do Express a cadeia de funções *middlware* de erro, e que instruem o sistema sobre o que fazer em caso de um evento de erro no servidor ser ativado.

Por último, são exportados o objeto Express e o *middleware* das sessões, de modo a que possam ser importados pelos outros ficheiros do servidor de web.

3.4.3 Routes, controladores e respostas do servidor

Recordando o que foi dito no Capítulo 2, os *routes* do Express são as funções que tratam dos pedidos HTTP do cliente, para um verbo HTTP específico e para um determinado URL. O servidor de web quando recebe o pedido, olha ao objeto *request*, que contém essas informações nas suas propriedades e distribui o pedido para o *route* correto. Quando um cliente envia um pedido HTTP que contenha o verbo GET, o objeto do *route* usa o método *.get()*. Nos casos em que o pedido seja feito com o verbo POST, o *route* evoca o método *post().* Desta forma, existirá uma função que recebe e processa o pedido HTTP do cliente para cada caminho relativo do website.

Todas as funções de *route* estão definidas no ficheiro JavaScript routes.js, presente na diretoria routes. Neste script, começa-se por se importar o módulo Express é criado o objeto express.Router(). É este objeto que tem acesso ao pedido do cliente e que determina a resposta que a este retorna. Enquanto que alguns pedidos são logo resolvidos neste script, para evitar que este fosse excessivamente longo, optou-se por se reencaminhar os pedidos para scripts individuais, denominados por controladores, onde é feito o processamento dos pedidos e consequentes respostas, tal como foi referido no capítulo anterior.

De um ponto de vista de mais alto nível, o processo de pedido-processamento-resposta que o *routing* da aplicação Express implementa pode ser visto no seguinte fluxograma:

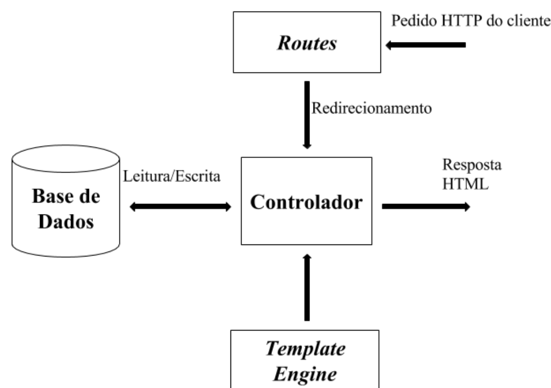

Figura 3.3 – Overview do processo de pedido-processamento-resposta implementado pelo Express. Adaptado de [35].

3.4.3.1 Página de índice

Quando um cliente introduz o URL https://santiago.co.it.pt/, correspondente à homepage do website, significa que é enviado um pedido HTTP GET ao servidor. Esse pedido pode ser visto na figura seguinte:

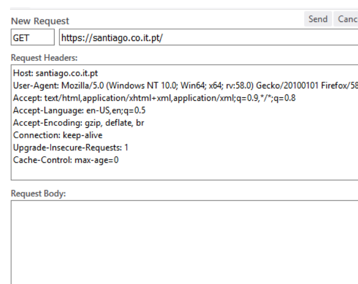

Figura 3.4 – Pedido HTTP GET enviado ao servidor

A aplicação Express recebe este pedido e distribui-o para a função router.get('/'). Esta função responde ao cliente gerando a página HTML index.html, através do *template engine* que se encontra na pasta views. No seu browser, o cliente recebe uma mensagem HTTP com o seguinte cabeçalho:

Response headers:

```
Server: nginx/1.10.3 (Ubuntu)
Date: Sun, 04 Feb 2018 03:54:12 GMT
Content-Type: text/html; charset=utf-8
Transfer-Encoding: chunked
Connection: keep-alive
X-Powered-By: Express
Cache-Control: no-cache, private, no-
store, must-revalidate, max-stale=0, post-
check=0, pre-check=0
ETag: W/"d5f-
pHXsqlTq2+Kcb82Ksr3By4zEdHA"
Content-Encoding: gzip
```

Figura 3.5 – Cabeçalho da resposta HTTP enviada ao cliente

A mensagem de resposta possui ainda um campo *body,* onde segue o código HTML que o browser interpreta e apresenta a página ao cliente.

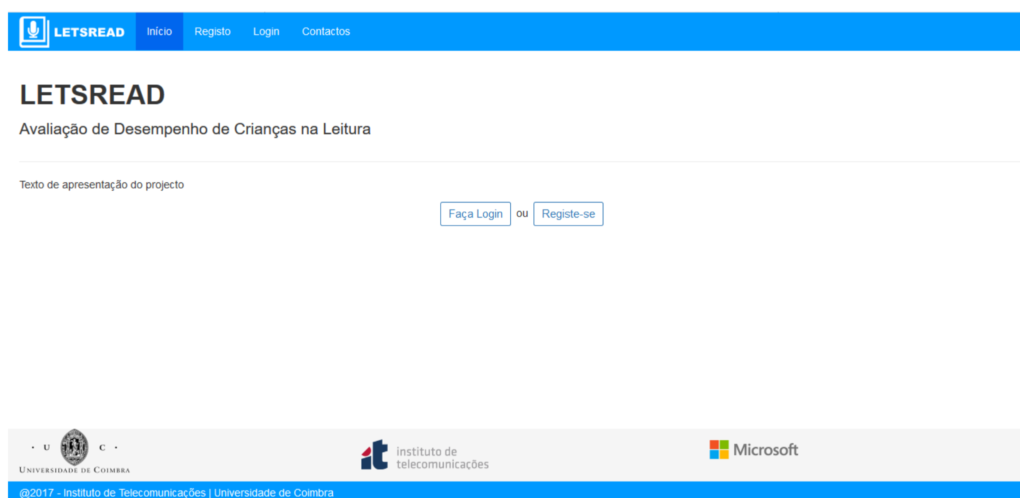

Figura 3.6 – Página de índice

A página pode ser seccionada em três partes distintas: *header*, corpo e *footer*. O *header* contém uma barra de navegação que permite aceder às páginas de registo, login e de contactos. No corpo da página, estão presentes dois botões que permitem aceder às páginas de registo de professores e de login. O *footer* da página contém hiperligações para três páginas externas.

3.4.3.2 Registo

Acedendo a https://santiago.co.it.pt/registo, o browser do utilizador envia um pedido GET, em tudo semelhante àquele apresentado na Figura 3.3, exceto no URL. A aplicação Express recebe essa mensagem, que é reencaminhada para função router.get('/registo'), que por sua vez reencaminha a mensagem para o registoController, o controlador criado para tratar dos pedidos feitos neste caminho relativo.

O controlador recebe o pedido GET, carrega o *template engine,* gera o código HTML e envia ao cliente a resposta HTTP ao cliente com os cabeçalhos ilustrados na figura 3.4 e com o código HTML da página no *payload* da resposta. O browser, por sua vez, olha ao código e apresenta a página de registo.

A página de registo permite a um professor efetuar o registo na plataforma. Apresenta um formulário com cinco campos de preenchimento obrigatório: nome completo, nome da escola onde leciona, e-mail, *username* e password.

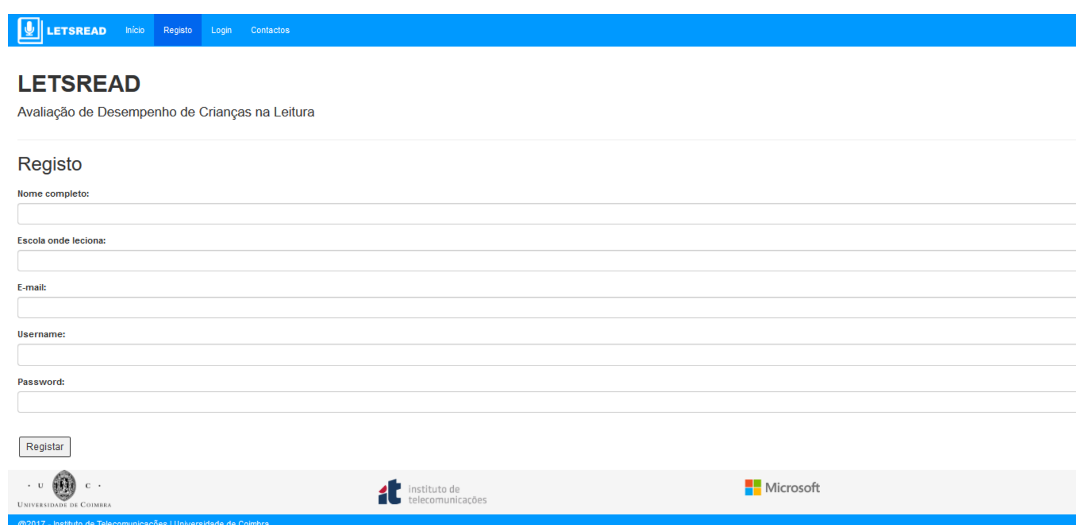

Figura 3.7 – Página de registo

Após o cliente introduzir os dados no formulário e clicar no botão Registar, o browser envia um pedido HTTP POST, encapsulando no seu *body* os dados introduzidos. Este pedido chega ao Express e é distribuído à função router.post('/registo'), que evoca a função registo_post do registoController.

Chegando a esta função, o controlador acede ao body do pedido do cliente, obtém os valores dos dados enviados e escreve-os na base de dados, numa nova linha da tabela Professores. Por omissão, o campo Autorizado da tabela Professores é preenchido com o

‘n’ (não autorizado). Antes de escrever na tabela a password enviada pelo cliente, o controlador evoca o módulo bcrypt[19] do Express que encripta esta *string*.

Caso este processo de escrita na base de dados tenha sido executado com sucesso, o servidor responde ao cliente com um alerta que o notifica que o processo de registo foi bem-sucedido, é redirecionado para a página de login e através do módulo do Node.js Node é enviado um email para o professor, providenciando-lhe algumas informações sobre o seu registo. Por outro lado, caso tenha acontecido algum problema no registo, como por exemplo o *username* introduzido, já constar na base de dados é enviada uma nova página HTML, onde o utilizador é alertado desse facto.

3.4.3.3 Login

Para o URL https://santiago.co.it.pt/login o processo repete-se. O browser envia o pedido HTTP GET, que é recebido pelo router.get(‘/login). Esta função reencaminha o ficheiro loginController.js. Este script, recebe os parâmetros do HTTP GET e responde ao cliente com a geração do *template engine* para a página do Login.

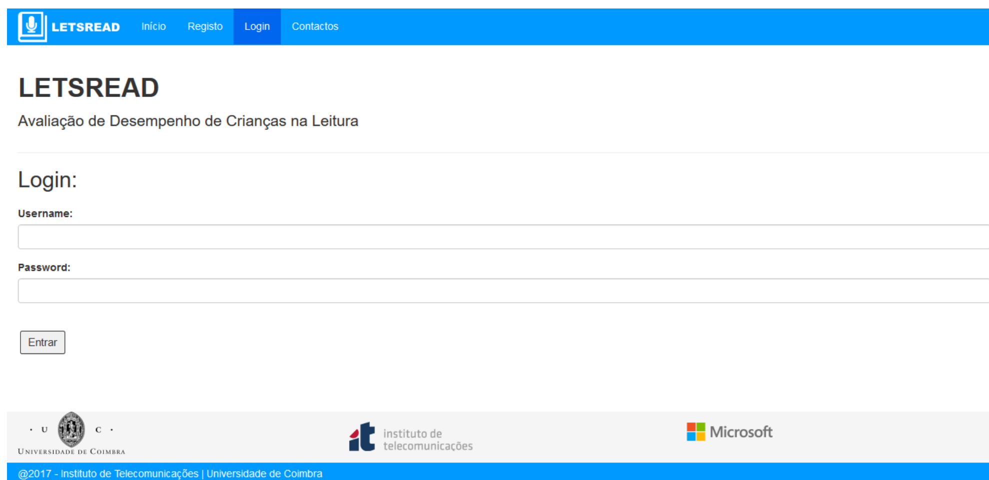

Figura 3.8 – Página de login no sistema

A página de login possibilita a entrada dos utilizadores registados ao sistema de avaliação de leitura. Contém um formulário com dois campos de preenchimento obrigatório: *username* e password. Carregando no botão Entrar, é endereçado um pedido HTTP POST para o servidor. A aplicação Express remete o pedido à função router.post(‘/login), que o reenvia à função login_post do *script* loginController. Aqui, começa por ser feita a verificação sobre a existência do *username* introduzido na tabela

[19] https://www.npmjs.com/package/bcrypt

Professores da base de dados. Caso o resultado seja negativo, o servidor de web envia a mesma página ao cliente, mas com uma mensagem de erro que informa que o *username* introduzido não consta nos registos. Se o resultado dessa verificação for positivo, é feita uma verificação ao nível da password. Se esta não corresponder à que está inserida nessa linha da base de dados, o servidor responde com a mesma página HTML, mas com uma mensagem de erro que alude a esse facto.

Quando o resultado de ambas as verificações é confirmado, o controlador cria, abre e guarda no ficheiro sessions uma sessão para esse cliente, através do *middleware* express-session. Para cada sessão, são guardadas duas variáveis de sessão: prof_id, que corresponde ao campo Prof_ID da tabela Professores da base de dados e prof_nome, coincidente com a entrada existente na coluna Nome da mesma tabela. É ainda guardado um cookie no browser do cliente onde é armazenado uma variável denominada por sid e que é composta por uma *string* identificadora, única para cada sessão. Por último, o servidor envia ao browser do cliente instruções para o redirecionar para a página https://santiago.co.it.pt/prof_index.

3.4.3.4 Prof_index

Quando o URL https://santiago.co.it.pt/prof_index é evocado pelo cliente, este envia uma mensagem HTTP GET, lida pela função router.get('/prof_index) da aplicação do Express, que de imediato evoca a função prof_index_get definida no *script* prof_indexController.js. Chegando aqui, é feita a averiguação se no cabeçalho do pedido HTTP estão presentes os campos referentes à sessão, nomeadamente o cookie com a informação do sid. Caso não seja encontrado este cookie de sessão, o servidor responde indicando ao browser que redirecione o cliente para a página de índice (https://santiago.co.it.pt/). Por outro lado, se o cookie de sessão constar do cabeçalho do pedido recebido, sinónimo de que o cliente tem autorização para aceder a esta página, é gerado o *template engine,* que faz a renderização de uma página HTML específica para cada cliente, uma vez que esta faz uso das variáveis de sessão acima mencionadas.

Esta página instrui o cliente sobre as funcionalidades que a plataforma permite fazer e apresenta hiperligações para outras áreas do site.

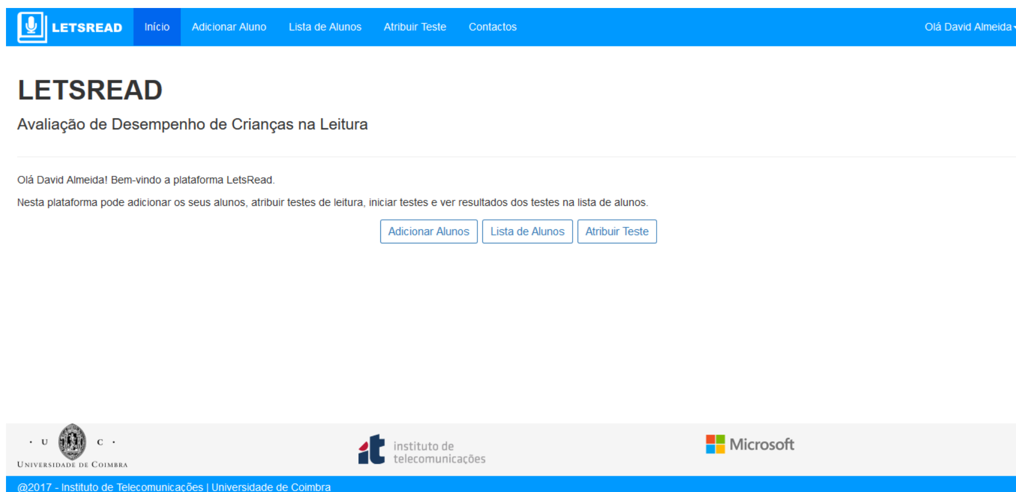

Figura 3.9 – Página de índice após login na plataforma LetsRead

3.4.3.5 Adicionar Alunos

Quando o professor acede à área do website com o caminho relativo do URL /adiciona, envia um pedido HTTP GET que depois de ser recebido pelo router.get('/adiciona') é passado à função aluno_create_get do *script* adicionaController.js.

Esta função, começa por averiguar se o cliente que enviou o pedido tem uma sessão ativa, através do cabeçalho do pedido. Caso isso não se verifique, a resposta enviada dá ordens ao browser para redirecionar o cliente para a página de índice.

Se existir sessão ativa, o servidor responde ao cliente enviando uma resposta HTTP com o código HTML da página *adiciona*, que é lhe é apresentada no browser.

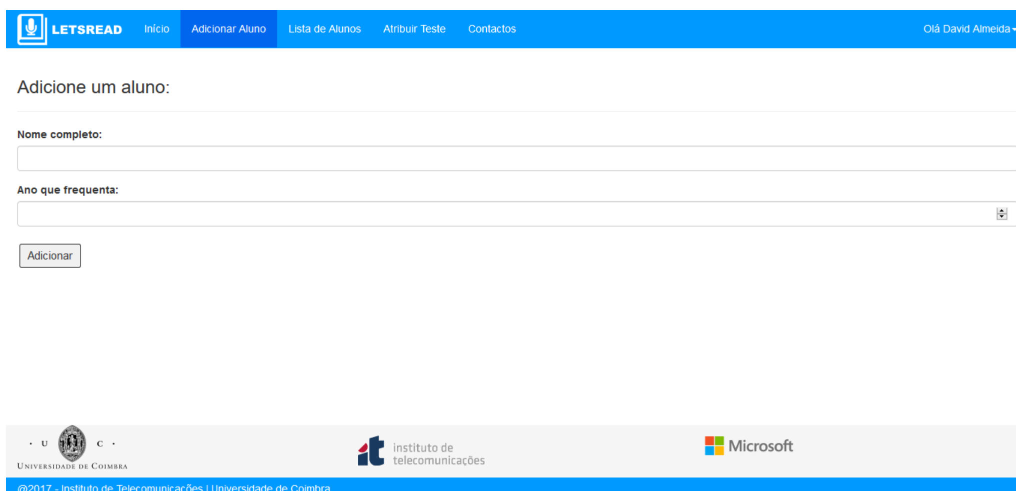

Figura 3.10 – Página para registo dos alunos

Esta página permite adicionar ao sistema os alunos a quem utilizador pretende atribuir testes. Contém um formulário com dois campos de preenchimento obrigatório: nome completo do aluno e ano que este frequenta. Este segundo campo apenas aceita como *input* um número entre um e quatro.

Ao clicar no botão carregar, o browser envia um pedido HTTP POST ao servidor, que o encaminha para o router.post('/adiciona'), que por sua vez o reencaminha para a função aluno_create_post, existente no *script* adicionaController.js. Esta função olha ao *body* do pedido, extrai os valores aí existentes e insere-os na tabela Alunos da base de dados, introduzindo igualmente, no campo ID_Professor, o valor da variável de sessão prof_id. Após este processo, o servidor de web responde ao cliente com o envio de uma página HTML que confirma a adição dos dados desse aluno ao sistema.

3.4.3.6 Lista de Alunos

Para aceder à página que permite ver a lista de alunos registados pelo cliente, o browser tenta aceder a https://santiago.co.it.pt/alunos e envia um pedido HTPP GET, recebido pela aplicação Express, que o distribui à função router.get('/alunos'). Esta, por sua vez, esta função evoca uma outra função, alunos_get, pertencente ao controlador alunosController.js. O pedido entra na função como argumento, através do objeto *req* e é feita a avaliação, através do cabeçalho do pedido recebido, sobre a existência de uma sessão ativa. Se esta avaliação não for bem-sucedida, o servidor envia uma resposta HTTP comunicando ao browser para redirecionar o cliente para a página de índice. Caso contrário, o servidor acede a esse campo, avalia o sid e através deste identificador obtém as variáveis de sessão prof_id e prof_nome.

Usando a variável prof_id, o controlador faz uma pesquisa à tabela de Alunos e retorna todos os resultados onde o campo Professor_ID seja igual ao que está guardado nessa variável de sessão. De seguida, é feita uma segunda pesquisa, cruzando a tabela Alunos com a tabela TestesAtribuidos, e que tem por objetivo a obtenção das informações sobre os testes atribuídos e/ ou realizados de cada um dos Alunos.

De seguida, é evocado o *template engine,* que gera uma página HTML dinâmica e onde estão incluídos os resultados dessas duas pesquisas à base de dados. O controlador conclui as suas tarefas, enviando uma resposta HTTP, para o browser do cliente, com essa página.

LETSREAD Início Adicionar Aluno Lista de Alunos Atribuir Teste Contactos Olá David Almeida

Lista de Alunos:

Nome	Ano	Teste	Último Teste
David António	4º	Atribuir Teste	Ver Último Teste
João Pedro	2º	Atribuir Teste	Ver Último Teste
Jorginho	4º	Atribuir Teste	-
Mario	1º	Atribuir Teste	Ver Último Teste

Universidade de Coimbra instituto de telecomunicações Microsoft

@2017 - Instituto de Telecomunicações | Universidade de Coimbra

Figura 3.11 – Página onde é feita a listagem dos alunos inseridos no sistema

Esta página lista numa tabela os alunos que o utilizador adicionou previamente. A tabela mostra a informação em quatro colunas diferentes: nome do aluno, com uma hiperligação para a página que contém o seu perfil pessoal; ano que o aluno frequenta; uma hiperligação para que seja feita uma atribuição de um teste a esse aluno ou uma hiperligação para a página onde é possível dar início à realização do teste, caso este já esteja previamente atribuído; uma hiperligação para a página de visualização dos resultados do teste mais recente que a criança tenha realizado.

3.4.3.7 Perfil do aluno

Para cada um dos alunos inseridos no sistema, é criado um URL, obtido a partir do identificador único existente na base de dados. O URL é composto a partir do caminho relativo */alunos.* A título de exemplo, para o aluno a que foi atribuído o identificador id=5 na coluna Aluno_ID da tabela Alunos, é criado o URL https://santiago.co.it.pt/alunos/5. O browser, acedendo a este caminho, envia um HTTP GET *request,* que é facultado ao router.get('/alunos/:id') e que reenvia o pedido para a função alunos_get_aluno do script alunosController.js. Após serem feitas as verificações sobre a existência de sessão, é avaliado se esse aluno está associado ao professor com a sessão ativa, através da coluna Professor_ID. Caso o pedido GET seja de um professor com a sessão ativa, mas o :id do URL a que tenta aceder não esteja associado ao seu identificador, o servidor de web responde-lhe com a indicação ao browser para o redirecionar para a página /prof_index. Se esse aluno estiver associado a esse professor, o controlador acede à base de dados e obtém os dados de todos os testes que esse aluno tenha realizado e/ou lhe tenha sido

atribuído, através de uma pesquisa cruzada entre as tabelas Alunos e TestesAtribuidos. Posteriormente, o controlador evoca o *template engine,* que renderiza e envia para o cliente a seguinte página HTML:

Figura 3.12 – Página pessoal do aluno

A página enviada apresenta os dados de dado aluno e lista o histórico de testes atribuídos. Cada teste no histórico tem associada uma hiperligação para a página de realização do teste (se a entrada na base de dados no campo Status da tabela TestesAtribuidos for “atribuído” ou “iniciado”), ou para página de visualização de desempenho de um teste (se o valor do status do teste for “realizado”). É ainda possível ao professor atualizar os dados sobre o nome e o ano do aluno. Se assim o fizer, é enviado um pedido HTTP POST, posteriormente processado pelo controlador, e é feito o *update* da entrada na tabela Alunos.

3.4.3.8 Atribuir Teste

Quando a aplicação Express recebe um pedido HTTP GET vindo do URL https://santiago.co.it.pt/atribui, evoca a função router.get(‘/atribui’). Esta, por sua vez, chama uma outra função, presente no ficheiro atribuiController.js, denominada por atribui_get().

Como esta é uma página restrita a professores com sessão ativa, o primeiro passo dessa função é fazer essa verificação. Se não existir sessão, a HTTP *response* do servidor contém instruções para que o browser do cliente faça o redireccionamento para a página de índice. No caso em que no cabeçalho do pedido consta o *cookie* de sessão, o controlador acede à variável de sessão prof_id e utiliza-a para fazer uma pesquisa à tabela

Alunos da base de dados, que retorna com as informações de todos os alunos associados ao professor. De seguida, é executada uma segunda pesquisa à base de dados, desta feita à tabela Testes, que devolve as colunas de todas as entradas nesta tabela.

Usando esses resultados, o controlador gera o código HTML a partir do *template engine* e envia ao cliente essa informação numa resposta HTTP. A página HTML inclui ainda um ficheiro JavaScript, forcebtn.js.

No corpo da página é apresentado um formulário com dois *dropdowns menus*. O da esquerda contém a lista de todos os alunos que o utilizador adicionou no sistema, enquanto que o menu da direita contém a listagem dos testes de leitura (previamente predefinidos) existentes na base de dados.

O cliente, ao escolher uma das opções de testes no menu *dropdown,* executa o ficheiro forcebtn.js, que envia um pedido HTTP POST, onde, no seu *body*, está incluída a informação sobre o identificador único do teste escolhido. Este *request* é recebido pelo controlador atribuiController.js, que faz uma pesquisa cruzada à base de dados entre a tabela Testes e a tabela Frases e, através dos identificadores, devolve as frases e pseudopalavras que constituem o teste. O controlador termina a sua tarefa gerando uma nova página HTML e enviando-a encapsulada numa resposta HTTP. O browser apresenta esta nova página ao cliente, que pode assim ver a constituição de cada um dos testes de leitura.

Figura 3.13 – Página de atribuição de um teste de leitura a um aluno.

Após o professor escolher um aluno e um teste de leitura, pode carregar no botão Atribuir, que envia um novo pedido HTTP POST para o servidor de web. A aplicação

Express recebe o pedido e entrega-o à função router.post('/atribui), que evoca a função atribui_post do controlador. Aqui, começa por ser feito uma pesquisa à tabela TestesAtribuidos, onde é verificado se para o aluno selecionado já existe uma entrada nessa tabela onde o campo relativo Status corresponde à *string* "atribuído". Se o resultado for afirmativo, essa entrada é atualizada e o valor da coluna Teste é trocado pelo valor que foi trazido pelo pedido HTTP. Para o caso contrário, é introduzida uma nova entrada na tabela TestesAtribuidos, definindo o Status como "atribuído". No fim deste processo, o controlador responde com uma nova página HTML, onde o professor é informado de que o teste foi atribuído com sucesso a esse aluno. Se assim o desejar, o professor pode imediatamente dar início ao teste de leitura, através de uma hiperligação existente nessa página.

3.4.3.9 Teste de Leitura

Tal como para o caso dos alunos, cada tarefa de leitura atribuída terá um URL único, formado através do identificador desse teste na base de dados. Desta forma, o URL desta área do website é dado por https://santiago.co.it.pt/tarefa/:id. Quando a aplicação do Express recebe um pedido HTTP GET deste caminho, o servidor encaminha-o para a função router.get('/tarefa/:id'), que por sua vez o remete para a função tarefa_get_tarefaatribuida. O controlador começa por averiguar a existência de uma sessão ativa nesse pedido. Se esta existir, é verificado se na tabela de TestesAtribuidos, o identificador da tarefa a que tenta aceder está associado ao identificador desse professor. Caso o resultado desta operação seja positivo, o passo seguinte é aferido o valor da coluna Status para entrada na tabela. Se esta tiver o valor de "atribuído" ou "iniciado" o controlador gera o código HTML da página de realização do teste, usando o *template engine*. Caso o campo Status contenha o valor "realizado", o controlador gera o código HTML da página de análise de resultados do teste, através do *template* ver_tarefa.

Quando é enviada página para realização de teste, no browser do cliente é apresentado o layout da [Figura 3.14].

Esta página carrega para o browser do cliente três scripts essenciais ao seu funcionamento: recorder_alt.js, socket.io.js (instalado pelo módulo Socket.io do Node.js) e testes_v2.js. O primeiro trata do manuseamento do Web Audio API, o segundo possibilita e gere o estabelecimento do *socket* com o servidor e o terceiro controla a lógica de envio e receção de mensagens através do *socket* e torna a página HTML dinâmica.

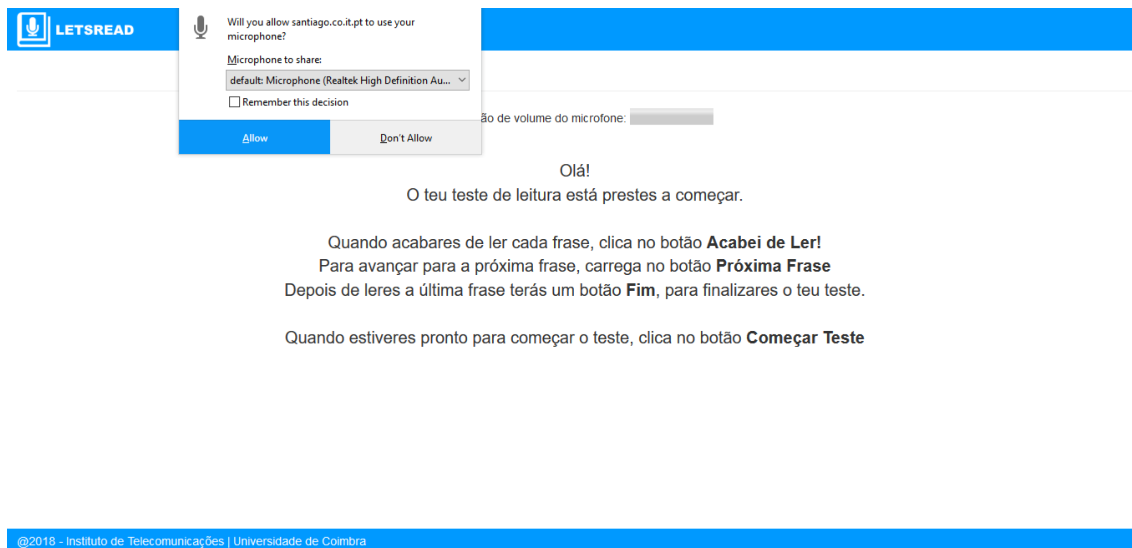

Figura 3.14 – Página de realização de um teste de leitura

Através da implementação do ficheiro socket.io.js, o browser começa o processo com o envio de um pedido HTTP GET especial onde pede ao servidor o estabelecimento de um *socket*. Esse canal de comunicação é estabelecido e todos os pedidos passam a ser transmitidos por essa via.

De seguida, é evocada a função mediaDevices.getUserMedia(), pedindo desta forma o acesso ao microfone do cliente. Se conceder permissão, é criado o objeto MediaStream, que contém o dispositivo de áudio referente ao microfone e envia esse objeto para ser processado pelo recorder_alt.js. Após a criação deste objeto, é evocada de 100 em 100 ms a função getVolume() do plugin recorder_alt. Os resultados retornados por esta função são usados para atualizar o vuímetro existente na página, onde o cliente pode perceber se o volume do áudio que o microfone recebe se encontra na região recomendável.

A partir deste momento, o processo de realização de um teste de leitura pode começar. A receção de frases e o envio de áudio para o servidor é controlado por eventos, tal como está ilustrado na seguinte figura:

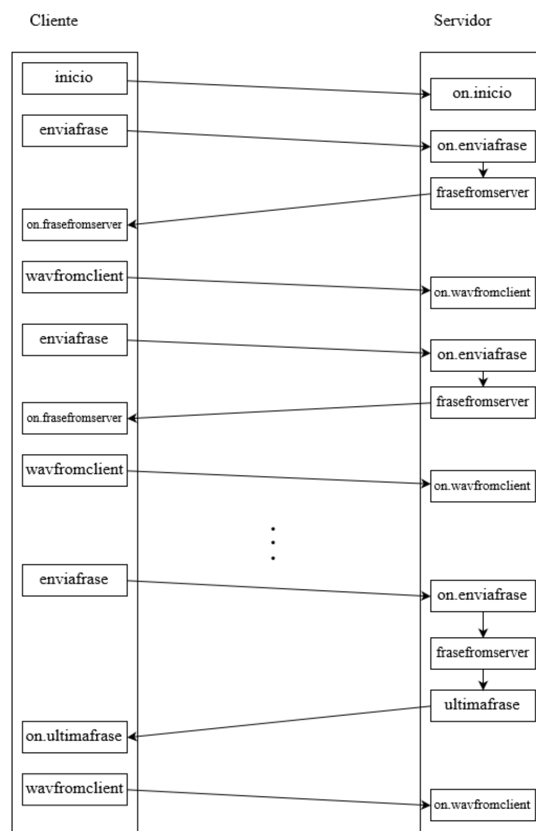

Figura 3.15 – Interações entre cliente e servidor, via socket, aquando a realização de uma tarefa de leitura.

Quando o aluno prime o botão Começar, é evocada o método stopVolume() do recorder_alt.js e são enviadas duas mensagens através dos eventos do *socket* inicio e enviafrase. O servidor, ao receber o evento on.inicio, atualiza o Status desse teste de "atribuído" para "iniciado" e armazena numa variável o identificador único do teste que está a ser iniciado. Quando o cliente envia a mensagem enviafrase, é acionado no servidor o evento on.enviafrase. Este *event handler* faz uma pesquisa à tabela Frases e retorna a primeira frase ou pseudopalavra desse teste e emite essa informação para o cliente numa mensagem, apelidada de frasefromserver. O browser recebe essa mensagem através do socket, começa a gravação de áudio apresenta a primeira frase que a criança terá de ler e troca o botão começar pelo botão Acabei de Ler. Quando se seleciona este botão, o browser aciona o evento wavfromclient, que pára a gravação de áudio e, através das funções do recorder_alt.js, envia o áudio para o servidor, que o recebe, em forma de *string* binária, na função de *event handler* on.wavfromclient.

Esta função começa por gravar o áudio num ficheiro em formato WAVE dentro de uma pasta, específica para cada professor, na diretoria audios. De seguida, uma nova entrada na tabela FrasesLidas é escrita, introduzindo todos os campos, exceto a coluna SegmentRes, que será adicionada mais tarde. O passo seguinte consiste em enviar o

ficheiro áudio para a função decodeWavFileAsync() do *addon* de reconhecimento de voz do Node.js, que retorna o valor de SegmentRes (uma *string* de regiões de áudio que correspondem à segmentação do sinal de fala em palavras ou disfluências) e que pode agora ser adicionado à base de dados. A análise destas regiões permite mais tarde avaliar o desempenho da leitura.

Na página do cliente, o botão “Acabei de Ler” é substituído pelo botão Próxima Frase, que, quando carregado, volta a ativar o evento enviafrase e o processo descrito é repetido ao longo do teste de leitura até que se chegue à ultima frase do teste. Neste caso, o servidor atualiza o Status para “realizado” e emite uma mensagem, com a *tag* ultimafrase, informando o browser do cliente de que o teste de leitura chegou ao fim e dando ordens para que o browser esconda o botão Próxima Frase e apresente o botão Fim. Carregando neste botão, é a apresentada a informação de que o teste de leitura chegou ao fim. A sessão do cliente é apagada e o browser é redirecionado para a página de login.

O servidor, por sua vez, evoca a função compute_score do script readingscore_v1.js, que faz a avaliação do teste, explicado em detalhe numa secção mais à frente.

Como já referido no início desta secção, quando o resultado da pesquisa inicial do controlador tarefaController.js resulta numa entrada com o Status “realizado”, este encaminha ao cliente uma resposta HTTP onde se inclui o código HTML gerado pelo *template* ver_tarefa.

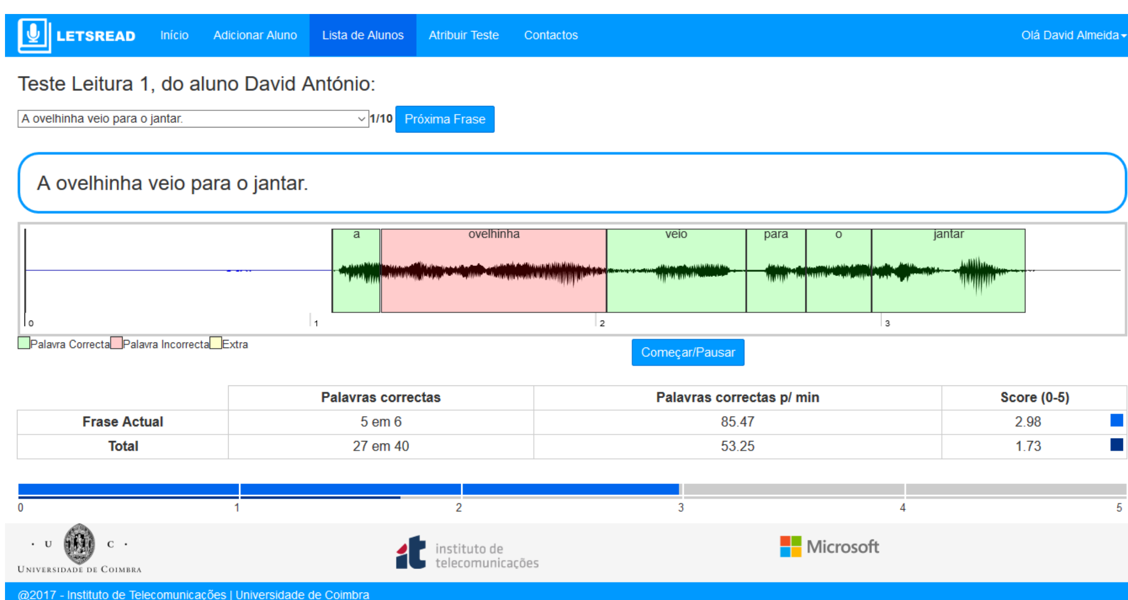

Figura 3.16 – Página de análise dos resultados de um teste de leitura.

É nesta página que o professor pode ver o desempenho dos alunos nos testes de leitura. Esta página é constituída por 5 secções distintas:

- Menu *dropdown* para selecionar a frase;
- Painel que apresenta a frase selecionada;
- Um *wavesurfer*, ferramenta onde o professor pode ouvir o áudio gravado e ver a sua forma de onda, bem como as regiões das palavras. Estas regiões podem estar pintadas de 3 cores: vermelho se a palavra foi considerada como incorretamente lida, amarelo se corresponder a um extra (hesitação ou uma pré articulação da palavra) ou verde, no caso em que a palavra foi tida como lida de maneira correta;
- Uma tabela que mostra a informação relativa ao número de palavras corretas, número de palavras totais, número de palavras corretas por minuto e o índice de leitura obtido. A primeira linha da tabela apresenta relativos à frase selecionada, ao passo que a segunda expõe os valores acumulados do teste completo.
- Duas barras que ilustram graficamente o índice de leitura obtido para a frase selecionada e para o acumulado do teste de leitura.

Para o bom funcionamento desta página, são ainda necessários outros 5 *scripts*: socket.io.js, ver_tarefa.js, para tornar a página dinâmica e definir os eventos do *socket* e 3 scripts para o *wavesurfer:* wavesurfer.min.js, wavesurfer.timeline.min.js, wavesurfer.regions.min.js.

Quando o browser carrega a página é estabelecido um *socket* entre o cliente e o servidor, e o emite, através dessa via, uma mensagem, onde inclui o identificador único da tarefa que pretende analisar. O servidor recebe essa informação e faz uma pesquisa cruzada entre as tabelas TestesAtribuidos e Frases, que retorna as frases e/ou pseudopalvras que constituem esse teste de leitura, o número de palavras corretas, o número de palavras totais, o número de palavras corretas por minuto e o índice de leitura obtido pela criança. Todos estes valores são emitidos para o cliente através do *socket* e ativam o evento on.prompts, que preenche a lista do menu de *dropdown,* a tabela das estatísticas acumuladas e a barra do índice de leitura total desse teste.

O professor quando seleciona uma dessas frases do menu de *dropdown* envia, através do *socket*, o identificador único da frase para o servidor. Este, executa uma pesquisa à tabela FrasesLidas e envia os resultados para o cliente. Além disso, é enviado o ficheiro de áudio numa *string* binária e os valores do número de palavras corretas, de

palavras totais, de palavras corretas por minuto e o índice de leitura obtido pela criança para essa frase. O browser, guarda o áudio num Blob, carrega-o para o *wavsurfer* e preenche a tabela e a barra do índice de leitura.

As interações entre servidor e cliente estão demonstradas na Figura 3.16:

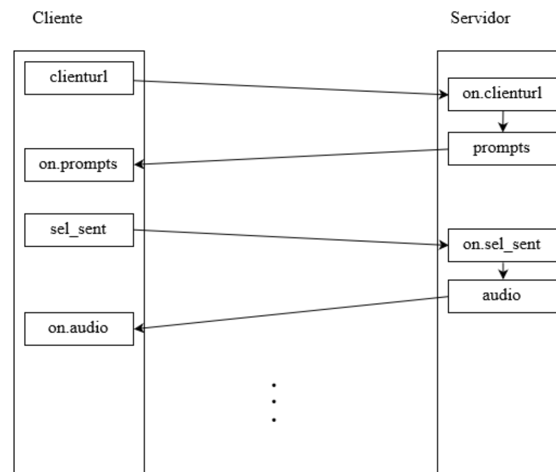

Figura 3.17 - Interações entre cliente e servidor, via socket, aquando a análise de uma tarefa de leitura.

3.4.3.10 Editar dados

Na eventualidade de um professor querer mudar os dados referentes ao seu registo, como por exemplo o seu endereço de e-mail ou a sua password, o website tem uma área que trata dessa funcionalidade, nomeadamente, no URL https://santiago.co.it.pt/editardados. O servidor avalia a existência de uma sessão ativa no cliente e, em caso afirmativo, envia uma resposta HTTP ao browser do cliente com a seguinte página de web:

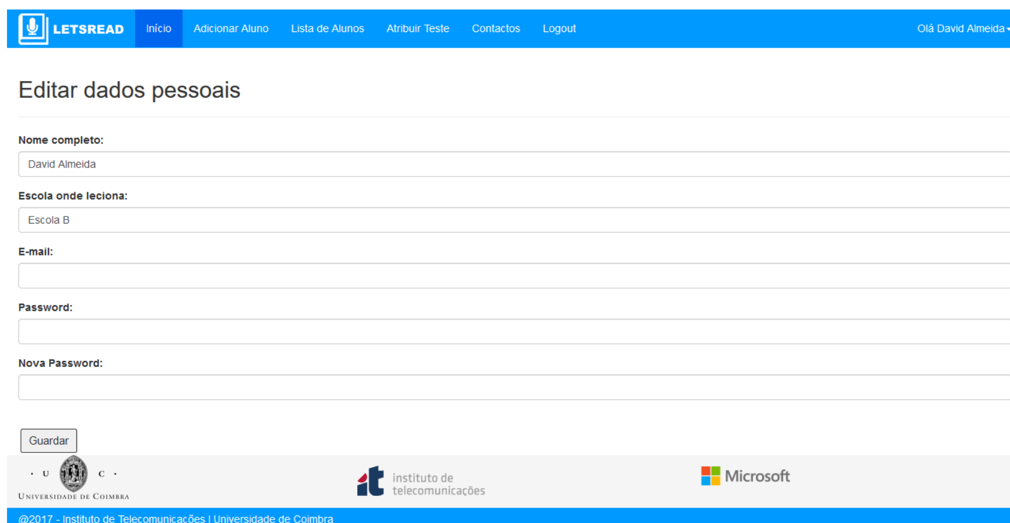

Figura 3.18 – Página para edição dos dados do professor.

Após introduzir os dados que quer alterar, o botão Guardar envia-os para o servidor, através de um pedido HTTP POST. O servidor faz uma pesquisa à tabela de Professores, atualiza os dados aí existentes por aqueles que chegam no pedido POST e responde com uma mensagem de alerta que notifica o cliente de que os dados foram atualizados com sucesso.

3.4.3.11 Logout

O cliente pode ainda aceder ao URL https://santiago.co.it.pt/logout. Quando assim o faz, envia um pedido HTTP GET à aplicação Express, que passa os controlos à função router.get('/logout'). Esta função reencaminha o pedido para o controlador logoutController.js. O controlador elimina todas as variáveis de sessão, o cookie de sessão e reencaminha o cliente para a página de índice.

3.4.4 Alterações ao recoder.js

Como foi avançado no Capítulo 2, foram realizadas algumas alterações ao plugin Recorder.js.

No seu código original, o Recorder.js recebe como input o objeto MediaStream, que contém os *tracks* de áudio provenientes do microfone do dispositivo do cliente e que os vai adicionando um buffer. O áudio capturado pelo microfone dos dispositivos dos clientes possui dois canais (esquerdo e direito) e utiliza por defeito uma frequência de amostragem de 48 kHz, codificado com 16 bits por amostra (2 bytes), os ficheiros WAV são enviados a esta frequência de amostragem. Houve assim, motivação para reduzir o tamanho dos ficheiros gravados, através de algumas modificações no código deste plugin. O ficheiro JavaScript que resulta das modificações efetuadas foi chamado de recorder_alt.js.

Ao invés de usar ambos os canais disponíveis, apenas é introduzido no buffer o áudio do canal esquerdo, que depois sofre um processo de decimação por 3. Além destas mudanças ao nível das amostras de áudio, foi necessário rever o cabeçalho WAV que as encapsula, de modo a que este seja coerente com as propriedades das amostras.

Antes da decimação das amostras de áudio, não foi aplicado nenhum filtro anti-aliasing, uma vez que em sinais de fala, muito raramente existem componentes de frequência acima dos 8kHz, o que justifica não serem filtrados, até por que isso

significaria um gasto acrescido dos recursos computacionais do cliente durante as gravações, o que não é de todo desejável num sistema com comunicação entre cliente e servidor em tempo real.

Assim, os ficheiros WAV decorrentes do uso do plugin adaptado têm áudio mono, com uma frequência de amostragem de 16 kHz, codificados com 16 bits (2 bytes) por amostra.

Fazendo uma comparação entre os ficheiros produzidos pelo recorder.js e recorder_alt.js, vem que para o primeiro, 1 segundo de gravação ocupa, sem o cabeçalho:

$$2\ (canal\ esquerdo\ e\ direito)\ \times 2\ (bytes) \times 48000 = 192\ 000\ bytes \quad (1)$$

Ao passo que para os ficheiros gerados pelo recorder_alt.js, 1 segundo de gravação possui, sem o cabeçalho:

$$1\ (canal\ esquerdo)\ \times 2\ (bytes) \times 16000 = 32\ 000\ bytes \quad (2)$$

Olhando aos resultados de (1) e (2), constata-se que se conseguiu uma redução em 83% no tamanho dos ficheiros de áudio.

Além das modificações ao nível do tamanho dos ficheiros em formato WAVE, foram adicionados dois novos métodos ao Recorder.js: *getVolume* e *stopVolume*. O método getVolume acede ao buffer com as amostras de áudio e que calcula o logaritmo da energia do sinal. Como a elaboração deste método partiu da necessidade em implementar um vuímetro na página de realização de um teste de leitura, de modo a que as transições dos níveis de volume apresentados no vuímetro sejam mais suaves, foi preciso incluir um filtro IIR de primeira ordem após o cálculo do logaritmo da energia do sinal, para introduzir um atraso na resposta. A função de transferência do filtro IIR pode ser vista na equação (3).

$$H(z) = \frac{1 - a}{1 - az^{-1}} \qquad (3)$$

O método stopVolume serve apenas para que os cálculos efetuados pela função getVolume sejam interrompidos.

3.4.5 Addon de reconhecimento de fala

Como anteriormente mencionado, o processamento dos sinais de áudio é feito através da função decodeWavFileAsync(), integrada num *addon,* escrito em C++, que foi desenvolvido no Laboratório de Processamento de Sinal do Departamento de Engenharia Eletrotécnica e de Computadores da Universidade de Coimbra.

Esta função recebe um ficheiro de áudio em formato WAVE, em conjunto com o texto da frase que era suposto ler. De seguida, o áudio é processado de maneira a que seja segmentado nas diferentes palavras e é detetado se a palavra foi correta ou incorretamente lida, bem como se existe conteúdo extra frase.

O algoritmo que compõe esta função é baseado em trabalho desenvolvido no laboratório acima referido [36] [37] e usa uma rede neuronal treinada com fala de crianças para descodificar o áudio em componentes acústicas [38]. É usada uma gramática de descodificação baseada nas sílabas da frase original, sendo no entanto permitido silêncio entre sílabas, repetições e pré-correções, de modo a se obter segmentos candidatos para palavras. Cada candidato é classificado como bem pronunciado ou não tendo em conta o rácio de verosimilhança entre a pronunciação ideal e o que foi reconhecido no áudio.

A função retorna para o servidor uma *string* de regiões, como a que é apresentada no seguinte exemplo:

```
"do:2.34:2.28:2;do:3.02:3.56:1;meu:3.56:3.93:1;tio:4.11:4.15:0"
```

A *string* de regiões é composta pelas palavras que compõem uma frase que a criança tenha lido, separadas por ponto e vírgula (;). Para cada uma das palavras, é ainda possível saber mais 3 campos, secionados pelo caracter :. Na secção imediatamente a seguir à palavra, é mostrado o tempo de inicio da locução dessa palavra, seguido pelo tempo de fim e por último, é apresentada uma *tag* que identifica se a leitura dessa palavra foi classificada como correta (1), incorreta (0) ou se representa algo extra frase (2).

3.4.6 Cálculo do índice de leitura

Para calcular o índice de leitura obtido num teste, são tomadas todas as *strings* de regiões de todas as frases e pseudopalavras proferidas pela criança durante a realização de um teste e é evocado o método compute_score, implementado no ficheiro readingscore_v1.

Este método recebe as *strings* de regiões e calcula os seguintes valores: Ncorr, correspondente ao número total de palavras corretas; Nwords, que representa o número total de palavras; T_dur é o valor da duração total das locuções. A partir dos valores de Ncorr e T_dur é calculado o número de palavras corretas por minuto, cwpm, através de

$$cwpm = \frac{Ncorr}{T_dur \times 60} \quad (4)$$

Apesar de existirem várias *features* para classificar a leitura de uma criança, neste projeto apenas foi utilizado o método que envolve o valor de cwpm, uma vez que este é um bom indicador da capacidade de leitura da criança [37]. É a partir deste valor que é calculado (ver equação 5) o índice de leitura, ou em inglês, *score* [37]. :

$$Score = 0.039063 \times cwpm - 0.35378 \; (5)$$

Após o cálculo do índice, este é armazenado na base de dados, nomeadamente no na coluna Score da tabela TestesAtribuidos.

3.4.7 Ngnix como *reverse proxy*

Apesar de o Node.js ser perfeitamente capaz de, por si só, funcionar como servidor de web, isso não é muito aconselhável, uma vez que, desta forma, o software que compõe a aplicação do servidor fica exposto a todos os pedidos HTTP vindos da internet, o que pode trazer algumas falhas de segurança e permitir ataques ao servidor [39]. De modo a mitigar esses possíveis problemas, foi usado um servidor reverse proxy, denominado por Nginx[20].

Um reverse proxy é um servidor intermediário, que reencaminha, para um outro servidor, os pedidos HTTP que recebe. Geralmente, este tipo de servidor serve de interface entre o software do servidor de web e a internet e introduz um nível adicional de abstração e controlo, assegurando uma comunicação entre servidor e cliente mais segura e fluida [40].

Desta forma, é possível correr o processo do servidor de web do Node.js numa rede local privada e é o reverse proxy que está em contacto direto com a internet.

[20] https://www.nginx.com/

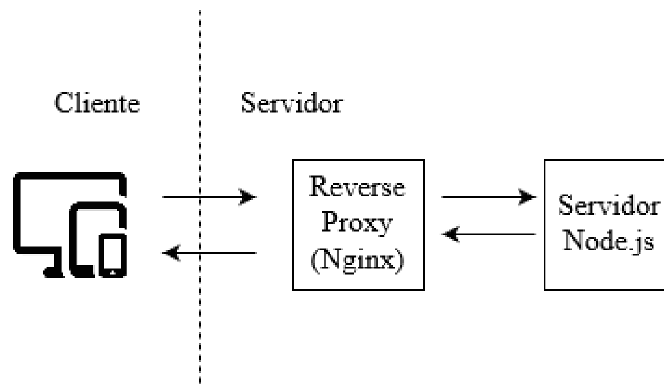

Figura 3.19 – Papel do reverse proxy. Adaptado de [41].

Além do trabalho que faz ao nível da segurança do servidor, foi igualmente importante usar o Nginx, pois este também permite gerir a utilização do protocolo TLS nas comunicações, essencial para que o sistema funcione através de HTTPS. Para isso, foi necessário incluir nos ficheiros de configuração do Nginx o caminho para os certificados SSL que foram atribuídos pela Autoridade de Certificação à máquina santiago.co.it.pt.

Capítulo 4

Testes ao sistema

Após o período de desenvolvimento do sistema LetsRead, o passo seguinte passou por serem feitos vários testes ao sistema. Foram testados vários pontos presentes nos standards de qualidade de software descritos pela ISO (International Organization for Standardization) [42] [43].

4.1 Testes de funcionalidade

No âmbito de Testes de Qualidade a Software, a funcionalidade refere-se à capacidade de o sistema desempenhar todas as funções a que este se propõe.

Foi testado o formulário da página de registo de professores e avaliou-se o cenário em que é tentado fazer um registo usando um *username* que já constasse na base de dados. Foi confirmado que o servidor envia uma mensagem de erro ao cliente, pedindo-lhe que escolhesse um *username* diferente.

Na página para login foram testados dois cenários: tentativa de autenticação com um *username* que não exista na base de dados e autenticação com um *username* que existe nos registos, mas cuja password não seja a que lhe está associada. Para ambos os casos, o sistema comportou-se da maneira que deveria e retornou ao cliente a respetiva mensagem de erro.

Foram igualmente feitos testes sobre permissões de acesso às páginas de perfil de aluno e de tarefas. Suponhamos que um professor, que estava autenticado no sistema, inseria no explorador do seu browser o URL https://santiago.co.it.pt/alunos/5, mas o aluno com o identificador id=5 não está associado a esse professor. O sistema prevê esses casos e instrui o browser do professor para reencaminhar para a página de índice. O mesmo processo acontece nas páginas de testes de leitura.

Um outro teste de funcionalidade que foi pertinente avaliar, prende-se com o preenchimento do formulário de edição dos dados referentes ao professor. Para os casos em que é pretendida mudar a password de acesso à plataforma é averiguado se a password atual corresponde àquela que consta nas entradas da base de dados e só é permitida a mudança se essa verificação for confirmada.

Outra característica que interessava testar foi a capacidade de o sistema funcionar corretamente em diferentes dispositivos, nomeadamente em PCs, telemóveis e tablets. Detetou-se que dispositivos com menores recursos de processamento (aconteceu especialmente em telemóveis), podem acontecer anomalias na gravação dos sinais de voz, causando que estes apareçam entrecortados, isto é, com algumas amostras com valor zero. Isto deve-se ao facto de o Web Audio API ser algo exigente em termos de poder computacional.

4.2 Testes de segurança

O capítulo da segurança do sistema LetsRead é de elevada importância, especialmente tendo em conta que são armazenados alguns dados sensíveis dos utilizadores. Portanto, foram feitos alguns testes que avaliaram esse aspeto. Foi usada principalmente a ferramenta Observatory by Mozilla[21]. Esta ferramenta faz uma série de pedidos HTTP ao servidor e analisa os cabeçalhos das respostas que o servidor retorna, especialmente ao nível do CSP (Content Security Policy). Este campo do cabeçalho controla a origem dos recursos carregados pela página de web (código HTML, imagens, ficheiros JavaScript, etc) e garante segurança contra cross-site scripting [44], isto é, ataques de injeção de *scripts* malignos no browser do cliente [45].

Num primeiro teste à segurança, a ferramenta Observatory avaliou o sistema com a nota F, a nota mais baixa que pode ser obtida. Isto motivou à inclusão do módulo do Node.js helmet, já abordado na secção 2.2.2. Este módulo implementa várias medidas de segurança ao nível dos cabeçalhos HTTP. Por omissão, o helmet não realiza nenhuma operação sobre o campo CSP, o que levou a uma alteração na configuração inicial desse módulo de modo a que sejam contempladas as medidas segurança do campo CSP. De seguida, foi realizado um novo teste com a ferramenta Observatory, que avaliou o sistema com A+, a classificação mais elevada que se pode obter. Os resultados deste teste podem ser vistos na figura 4.1.

[21] https://observatory.mozilla.org/

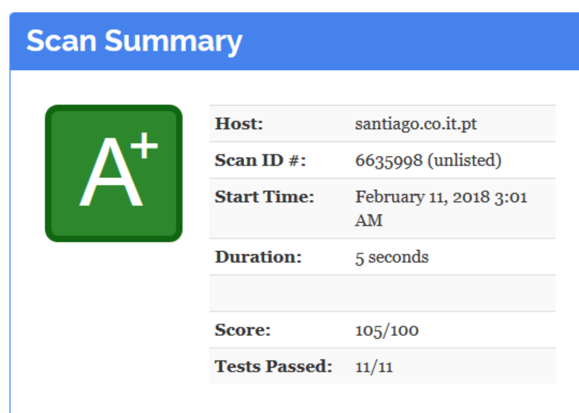

Figura 4.1 – Resultados do teste de segurança

4.3 *Feedback* dos utilizadores

Terminada a fase de desenvolvimento do software do sistema, partiu-se para o teste em ambiente de produção. Foi pedido a vários pais que fizessem o papel de professores e atribuíssem alguns testes de leitura às crianças. Foi igualmente requisitado o preenchimento de um pequeno questionário para atestar a satisfação dos utilizadores e o desempenho do sistema. As primeiras 5 questões abordadas foram adaptadas das regras ditadas pela System Usability Scale (SUS), que consiste em inquirir os utilizadores, sob a forma de uma escala linear entre 1 e 5 [46]. Foram ainda feitas algumas questões de resposta binária (apenas possível responder "Sim" ou "Não") e um espaço para comentários críticos. A lista de questões que constituem o inquérito, pode ser encontrada no Anexo B.

Como referido, as primeiras 5 questões seguem a escala SUS e como tal, os dados das respostas foram tratados através das fórmulas de cálculo impostos por esta escala [46]. Assim, para estas questões foi obtido, em média, uma pontuação de 83.75 pontos em 100 possíveis.

Para as restantes 3 questões de resposta binária obtiveram-se, em média, uma percentagem de 100% na resposta "Sim".

Os resultados do feedback dos testadores foram positivos, apesar de à data da entrega desta dissertação, a amostra obtida não permitir extrapolar resultados que demonstrem um panorama mais geral sobre a satisfação dos utilizadores da plataforma.

Capítulo 5
Conclusão

O principal objetivo desta dissertação era implementar um sistema online e em tempo real, que fosse capaz de servir coma plataforma para os professores, pais e tutores avaliarem a capacidade de leitura de crianças com idades compreendidas entre os 6 e os 10 anos. Esse objetivo foi atingido e os indicadores de satisfação dos utilizadores que testaram o sistema foram francamente positivos.

Verificou-se que a escolha do Node.js foi certeira, por permitir o adicionamento de vários módulos que tratam das diferentes tarefas, simplificando o processo de programação do código do servidor. A oferta de suporte a módulos que tratam do estabelecimento e gestão de *sockets* foi crucial para que fosse possível haver comunicação em tempo real entre browser e servidor.

O Web Audio API também demonstrou ser uma ferramenta extremamente útil, mas com o senão de ser exigente em poder computacional, o que pode causar algumas anomalias nos sinais de fala captados pelos dispositivos com menor capacidade de processamento. Um possível trabalho futuro pode passar por encontrar uma maneira de gravar ficheiros em formato WAVE sem ser necessário o uso do Web Audio API.

A nível pessoal, trabalhar neste projeto foi muito desafiante e bastante gratificante, dada a utilidade que este sistema tem e por ser uma ferramenta que pode auxiliar a melhoria da capacidade de leitura de crianças portuguesas. O âmbito desta dissertação significou o estudo de múltiplas tecnologias usadas em ambiente de *web development,* uma área que desperta particular interesse no autor.

Bibliografia

[1] "Wikipedia Cliente-servidor." [Online]. Available: https://pt.wikipedia.org/wiki/Cliente-servidor. [Accessed: 09-Feb-2018].

[2] "What is a web server? - Learn web development | MDN." [Online]. Available: https://developer.mozilla.org/en-US/docs/Learn/Common_questions/What_is_a_web_server. [Accessed: 12-Jan-2018].

[3] "An overview of HTTP - HTTP | MDN." [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview. [Accessed: 13-Jan-2018].

[4] "Wikipedia: List of HTTP header fields." [Online]. Available: https://en.wikipedia.org/wiki/List_of_HTTP_header_fields#Request_fields. [Accessed: 13-Jan-2018].

[5] "Wikipedia: List of HTTP status codes." [Online]. Available: https://en.wikipedia.org/wiki/List_of_HTTP_status_codes. [Accessed: 13-Jan-2018].

[6] "Wikipedia: Response fields." [Online]. Available: https://en.wikipedia.org/wiki/List_of_HTTP_header_fields#Response_fields. [Accessed: 13-Jan-2018].

[7] "Wikipedia: Certificate authority." [Online]. Available: https://en.wikipedia.org/wiki/Certificate_authority. [Accessed: 13-Jan-2018].

[8] T. Hughes-Croucher and M. Wilson, *Node Up and Running*. O'Reilly Media, 2012.

[9] "Node.js State of the Union 2017 – Node.js Collection – Medium." [Online]. Available: https://medium.com/the-node-js-collection/node-js-state-of-the-union-blog-2017-ed86640ec451. [Accessed: 11-Jan-2018].

[10] "Stack Overflow Developer Survey 2017." [Online]. Available: https://insights.stackoverflow.com/survey/2017#technology. [Accessed: 20-Jan-2018].

[11] "Node.js." [Online]. Available: https://nodejs.org/en/. [Accessed: 08-Jan-2018].

[12] P. Teixeira, *Professional Node.js - Building Javascript Based Scalable Software*. 2013.

[13] "The Node.js Event Loop, Timers, and process.nextTick() | Node.js." [Online]. Available: https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/. [Accessed: 09-Jan-2018].

[14] "npm." [Online]. Available: https://www.npmjs.com/. [Accessed: 09-Jan-2018].

[15] Evan M. Hahn, *Express in Action*. Manning, 2016.

[16] "Express 4.x - API Reference." [Online]. Available: http://expressjs.com/en/4x/api.html. [Accessed: 10-Jan-2018].

[17] "Express error handling." [Online]. Available: http://expressjs.com/en/guide/error-handling.html. [Accessed: 12-Jan-2018].

[18] "Express routing." [Online]. Available: http://expressjs.com/en/guide/routing.html. [Accessed: 11-Jan-2018].

[19] "API Reference – Pug." [Online]. Available: https://pugjs.org/api/reference.html. [Accessed: 11-Jan-2018].

[20] "Using template engines with Express." [Online]. Available: https://expressjs.com/en/guide/using-template-engines.html. [Accessed: 11-Jan-2018].

[21] "GitHub - mapbox/node-sqlite3: Asynchronous, non-blocking SQLite3 bindings for Node.js." [Online]. Available: https://github.com/mapbox/node-sqlite3. [Accessed: 12-Jan-2018].

[22] M. Owens and G. Allen, *The definitive guide to SQLite*. Apress, 2010.

[23] "Zero-Configuration." [Online]. Available: https://www.sqlite.org/zeroconf.html. [Accessed: 13-Jan-2018].

[24] "Query Language Understood by SQLite." [Online]. Available: https://www.sqlite.org/lang.html. [Accessed: 12-Jan-2018].

[25] R. Rai, *Socket.IO Real-time Web Application Development*. Packt Publishing,

2013.

[26] "TCP Definition." [Online]. Available: http://www.linfo.org/tcp.html. [Accessed: 15-Jan-2018].

[27] "Socket.IO — Client API." [Online]. Available: https://socket.io/docs/client-api/#. [Accessed: 14-Jan-2018].

[28] "MediaStream - Web APIs | MDN." [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/MediaStream. [Accessed: 13-Jan-2018].

[29] "Blob - Web APIs | MDN." [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Blob. [Accessed: 15-Jan-2018].

[30] B. Smus, *Web Audio API*. 2013.

[31] "Using the Web Audio API - Web APIs | MDN." [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API/Using_Web_Audio_API. [Accessed: 15-Jan-2018].

[32] "Web Audio API." [Online]. Available: https://webaudio.github.io/web-audio-api/. [Accessed: 16-Jan-2018].

[33] "Microsoft WAVE soundfile format." [Online]. Available: http://soundfile.sapp.org/doc/WaveFormat/. [Accessed: 16-Jan-2018].

[34] "05 - Working with package.json | npm Documentation." [Online]. Available: https://docs.npmjs.com/getting-started/using-a-package.json. [Accessed: 20-Jan-2018].

[35] "Express Tutorial Part 4: Routes and controllers - Learn web development | MDN." [Online]. Available: https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/routes. [Accessed: 18-Jan-2018].

[36] J. Proença, C. Lopes, M. Tjalve, A. Stolcke, S. Candeias, and F. Perdigão, "Detection of mispronunciations and disfluencies in children reading aloud," in *Proceedings of the Annual Conference of the International Speech Communication Association, INTERSPEECH*, 2017, vol. 2017–Augus, pp. 1437–

1441.

[37] J. Proença, C. Lopes, M. Tjalve, A. Stolcke, S. Candeias, and F. Perdigão, "Automatic evaluation of reading aloud performance in children," *Speech Commun.*, vol. 94, pp. 1–14, 2017.

[38] J. Proença, D. Celorico, S. Candeias, C. Lopes, and F. Perdigão, "The LetsRead Corpus of Portuguese Children Reading Aloud for Performance Evaluation," in *ELRA International Conf. on Language Resources and Evaluation, - LREC*, 2016, pp. 781–785.

[39] "5 Tips to Increase Node.js Application Performance." [Online]. Available: https://www.nginx.com/blog/5-performance-tips-for-node-js-applications/. [Accessed: 08-Feb-2018].

[40] "What is a Reverse Proxy Server? | NGINX." [Online]. Available: https://www.nginx.com/resources/glossary/reverse-proxy-server/. [Accessed: 08-Feb-2018].

[41] "Setup Node.js, Apache and an nginx reverse-proxy with Docker." [Online]. Available: https://medium.com/@francoisromain/setup-node-js-apache-nginx-reverse-proxy-with-docker-1f5a5cb3e71e. [Accessed: 08-Feb-2018].

[42] "ISO/IEC 25010:2011 - Systems and software engineering -- Systems and software Quality Requirements and Evaluation (SQuaRE) -- System and software quality models." [Online]. Available: https://www.iso.org/standard/35733.html. [Accessed: 11-Feb-2018].

[43] "ISO 9241-210:2010 - Ergonomics of human-system interaction -- Part 210: Human-centred design for interactive systems." [Online]. Available: https://www.iso.org/standard/52075.html. [Accessed: 11-Feb-2018].

[44] "Security/Guidelines/Web Security - MozillaWiki." [Online]. Available: https://wiki.mozilla.org/Security/Guidelines/Web_Security#Content_Security_Policy. [Accessed: 11-Feb-2018].

[45] "Cross-site Scripting (XSS) - OWASP." [Online]. Available: https://www.owasp.org/index.php/Cross-site_Scripting_(XSS). [Accessed: 11-Feb-2018].

[46] "MeasuringU: Measuring Usability with the System Usability Scale (SUS)." [Online]. Available: https://measuringu.com/sus/. [Accessed: 11-Feb-2018].

[47] "Client-Server overview - Learn web development | MDN." [Online]. Available: https://developer.mozilla.org/en-US/docs/Learn/Server-side/First_steps/Client-Server_overview. [Accessed: 13-Jan-2018].

[48] "Cloud Server SSL Price Reduction | Domainsrush Domain Registration." [Online]. Available: https://domainsrush.com/cloud-server-ssl-price-reduction/. [Accessed: 07-Feb-2018].

[49] "WebSocket | PubNub." [Online]. Available: https://www.pubnub.com/learn/glossary/what-is-websocket/. [Accessed: 14-Jan-2018].

Anexo A

Estrutura da Base de Dados

Como referido no Capítulo 3, foi necessário gerar uma base de dados onde estarão armazenados todos os dados necessários para o funcionamento do sistema. Os dados estão distribuídos pelas 6 tabelas descritas de seguida.

- Tabela Professores.
  Armazena os dados relativos aos professores que se registem no sistema. Contém as seguintes colunas:
  - Prof_ID. Uma variável do tipo INT, que serve de identificador único para cada professor registado no sistema. É a chave primária da tabela.
  - Escola. Campo onde consta o nome da escola onde o professor leciona.
  - Nome. Coluna onde estará registado o nome do professor.
  - Username. Uma *string* que identifica cada professor registado. É usada essencialmente para efeitos de autenticação.
  - Email. Campo destinado ao armazenamento do endereço de email de cada professor.
  - Password. Esta coluna armazena a password de cada professor. É usada no ato de autenticação no website. A entrada na base de dados é composta por uma *string* encriptada, de modo a salvaguardar a privacidade dos utilizadores.
  - Autorizado. Este campo contém uma tag que indica se o registo do professor está autorizado a fazer login. Pode conter 2 valores: ‘n’ para os casos em que o registo ainda não verificado pelo administrador do sistema ou ‘s’ caso o registo tenha sido validado.
- Tabela Alunos.
  Contém os dados sobre os alunos que cada professor adiciona ao sistema. Esta tabela possui 4 colunas. Relaciona-se com a tabela Professores.
  - Aluno_ID. Um número inteiro identificador de cada aluno introduzido na base de dados. É a chave primária da tabela.
  - Nome. *String* que contém o nome do aluno.

- Ano. Número inteiro compreendido entre 1 e 4, correspondente ao ano letivo que o aluno frequenta.
- ID_Professor. Corresponde ao Prof_ID (chave primária da tabela Professores) do professor que registou esse aluno. É através desta igualdade de colunas que ambas as tabelas estabelecem uma relação.

- Tabela Frases

  Nesta tabela estão guardados os dados referentes às frases e pseudopalavras que serão usadas para avaliar a leitura das crianças. Inclui os seguintes campos:

  - ID_Frase. Corresponde a um número inteiro identificador de cada frase introduzida na base de dados. É a chave primária da tabela.
  - Frase. Esta coluna conterá a frase ou pseudopalavra.
  - PP. Este campo corresponde a uma *flag* que determina se o texto inserido na coluna Frase é uma frase ou uma pseudopalavra. Esta coluna apenas pode tomar dois valores: *s* se o texto for uma pseudopalavra ou *n* se o texto for uma frase.

- Tabela FrasesLidas

  Esta tabela conterá a informação sobre todas as frases lidas durante a realização dos testes, de todos os alunos introduzidos no sistema. Relaciona-se com as tabelas Alunos, Frases e TestesAtribuidos.

  - ID_TRea. Corresponde ao campo TA_ID (chave primária da tabela TestesAtribuidos), que identifica a que teste pertence essa frase lida.
  - ID_Alu. É o identificador único (chave primária da tabela Alunos) do aluno que leu essa frase ou pseudopalavra.
  - NumFrase. Constituida pelo identificador ID_Frase da tabela Frases, que é a chave primária da tabela Frases. É a partir deste campo que é possível saber qual foi o texto que a criança leu.
  - SegmentRes. Armazena o resultado retornado pelo reconhecedor implementado no addon (segmentação de palavras com indicação se estão corretamente pronunciadas).
  - NomeAudio. Regista o caminho completo da diretoria onde o ficheiro de áudio gravado durante a leitura dessa frase foi escrito.

- Tabela Testes

  Esta tabela permite saber os dados que envolvem a constituição de um teste de leitura. Esta tabela está relacionada com a tabela Frases. Possui as colunas seguintes:

  - Teste_ID. Número inteiro, identificador único de cada teste. É a chave primária da tabela.
  - NumFrases. Inteiro que informa o número de frases e/ou pseudopalavras que compõem cada teste.
  - Frases. *String* que integra os identificadores das frases (as chaves primárias da tabela Frases) que constituem cada teste.
  - Str_ID. O nome dado a cada teste.

- Tabela TestesAtribuidos

  É nesta tabela que estão registados os todos testes de leitura que são atribuídos aos alunos. Relaciona-se com a tabela Alunos e Testes.

  - TA_ID. Número inteiro, identificador único de cada teste atribuído. É a chave primária da tabela.
  - Teste: Corresponde ao Teste_ID, identificador único para cada entrada da tabela Testes.
  - Alun_ID. Refere-se ao identificador único do aluno (ID_Aluno da tabela Alunos) que efetua esse teste de leitura.
  - Score. Refere-se ao resultado do cálculo do índice de leitura que a criança obteve para esse teste. É um número compreendido entre 0 e 5.
  - Status. Este campo permite saber o estado atual de cada teste. Apenas pode assumir três estados: 'atribuido', caso o teste já esteja atribuído mas a criança ainda não iniciou o teste de leitura; 'iniciado', caso a criança esteja a realizar o teste de leitura; 'realizado', quando a criança já tenha concluído a realização de um teste de leitura.
  - DataIni. Este campo regista a data e hora do início da realização do teste de leitura.

- DataFim. Guarda a data e hora do fim da realização do teste de leitura.
- Ncorr. Armazena um número inteiro, representante do total de palavras corretas que a criança obteve na realização do teste de leitura.
- Nwords. Regista o número total de palavras que a criança leu durante a realização do teste de leitura.
- Cwpm. Nesta coluna é guardado o número de palavras corretas por minuto que a criança articulou durante o seu teste de leitura.

Anexo B

Inquérito aos testadores da plataforma LetsRead

Numa escala de 1 a 5, como avalia o website em termos de navegabilidade (facilidade em chegar a qualquer área do website)? *

	1	2	3	4	5	
Muito Insatisfeito	○	○	○	○	○	Muito Satisfeito

Numa escala de 1 a 5, como avalia o design do website? *

	1	2	3	4	5	
Muito Insatisfeito	○	○	○	○	○	Muito Satisfeito

Numa escala de 1 a 5, como avalia a rapidez de resposta do website? *

	1	2	3	4	5	
Muito Insatisfeito	○	○	○	○	○	Muito Satisfeito

A experiência do utilizador (professor) foi intuitiva? *

	1	2	3	4	5	
Pouco Intuitiva	○	○	○	○	○	Muito Intuitiva

A experiência do utilizador (aluno) foi intuitiva? *

	1	2	3	4	5	
Pouco Intuitiva	○	○	○	○	○	Muito Intuitiva

Durante a realização de um teste, foram claros os momentos em que estava a ser feita a gravação do áudio? *

○ Sim

○ Não

Durante a análise de um teste de leitura, o áudio das locuções correspondeu às frases apresentadas? *

○ Sim

○ Não

Durante a análise de um teste de leitura, ficou claro quais as palavras consideradas corretas, quais as consideradas incorretas e qual o índice de leitura obtido nesse teste? *

○ Sim

○ Não

O seu feedback é precioso. Se tiver sugestões para melhorar o website ou algum erro a reportar, por favor indique neste campo.

Your answer