universidade de aveiro
theoria poiesis praxis

Projeto 2 - Relatório Final

Abel Fernandes Neto

Daniel Azevedo Alves

Diogo Daniel Soares Ferreira

Luís Davide Jesus Leira

Universidade de Aveiro - Laboratórios de Informática - Grupo 3

7 de Junho de 2015

Conteúdo

Lista de Figuras

Parte I

Apresentação

Capítulo 1

Resumo

Este relatório consiste na identificação dos objetivos propostos, na análise à estrutura da aplicação, na descrição da implementação efetuada e nas conclusões finais sobre a solução implementada. Encontram-se em anexo os testes realizados à aplicação.

O objetivo deste projeto é desenvolver uma aplicação *web* capaz de replicar o funcionamento de um órgão de *Hammond* (Figura 1.1). Através de pautas em formato Ring Tone Text Transfer Language (*RTTTL*), a aplicação deverá ser capaz de reproduzir o respetivo som, de acordo com um registo passado pelo utilizador e de efeitos adicionais opcionais. A aplicação é composta por cinco módulos principais: a interface *Web*, a aplicação principal, o sintetizador, o interpretador de pautas e o processador de efeitos. O repositório utilizado para o desenvolvimento da aplicação está alojado no *CodeUA* e o seu nome é "labi2015-proj2-g3".

Figura 1.1: Órgão de *Hammond*. Fonte: Wikipédia

Parte II

Desenvolvimento

Capítulo 2

Estrutura

Uma aplicação *Web* baseia-se numa página web, normalmente apresentada num *browser*, onde o utilizador poderá executar alguma tarefa para além de apenas ler o conteúdo da página. O utilizador poderá interagir com a página, a qual se encarrega de efetuar pedidos ao servidor, o qual tem como objetivo ser a base da aplicação, efetuando as respostas aos pedidos realizados e que transmite ao utilizador (caso seja necessário). As aplicações *Web* devem visar pela sua simplicidade e pelo cumprimento de tarefas de forma direta e eficaz.

A interface *Web* encontra-se disponível numa versão *desktop* e numa versão *mobile*. O código *JavaScript* considera-se essencial em ambas as versões. A interface deverá permitir ao utilizador criar músicas e versões das mesmas (podendo alterar os efeitos e os registos aplicados inicialmente aquando da criação da música), listar as músicas e versões existentes, aceder a informações básicas de todas as versões das músicas criadas, obter um gráfico das notas, poder ouvir cada versão através de um ficheiro áudio e atribuir votos positivos ou negativos a cada versão. A interface *Web* deverá comunicar com o servidor através de mensagens no formato JavaScript Object Notation (*JSON*).

A aplicação principal tem como módulo principal o *Cherrypy*. Deverá assegurar que responde corretamente aos pedidos realizados pelo utilizador da aplicação. Também deve criar um gráfico de notas quando pedido (através do módulo de *python matplotlib*) e disponibilizar ficheiros áudio que o utilizador poderá reproduzir de cada versão. Este módulo irá comunicar também com uma base de dados (através do módulo de *python sqlite3*) para guardar cada música criada e suas versões, tal como todas as informações associadas.

O interpretador de pautas deverá receber uma pauta em formato *RTTTL* e analisar o seu conteúdo. Se for inválido, deverá lançar um erro e devolver uma lista vazia. Caso contrário, deverá devolver à aplicação principal uma lista de pares (duração, frequência), de acordo com a pauta recebida e as frequências fundamentais.

O sintetizador deverá, de acordo com os pares (duração, frequência) recebidos através do interpretador, e de um certo registo, criar uma onda com a frequência

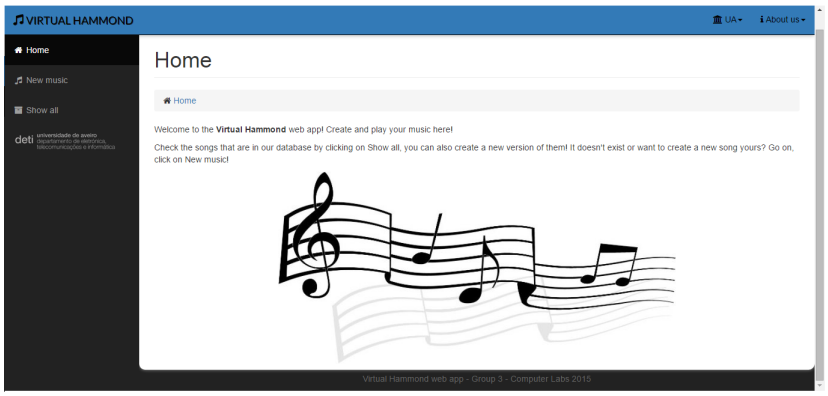

Figura 2.1: Página *Home* da aplicação para *desktop*.

correta e cada registo introduzida. Depois, deverá somar todas as ondas criadas, de forma a criar uma música, e enviar esses dados para o processador de efeitos, retornando-o.

O processador de efeitos deverá, após receber os *samples* da onda recebidos através do sintetizador, aplicar os efeitos pedidos pelo utilizador (serão descritos posteriormente) e criar a música num diretório específico para armazenar as músicas. Depois, deverá retornar o diretório onde a música se encontra.

2.1 Interface Web

A interface *web* é apresentada sob uma versão *desktop* e uma versão *mobile*. Para primeira foi usado um *template Bootstrap* (nomeadamente, *SB Admin* `http://startbootstrap.com/template-overviews/sb-admin/`, tendo como função ser uma base inicial bastante primária, tendo sido posteriormente trabalhada e desenvolvida de encontro com o que era pretendido). Para a interface *mobile* foi utilizado *Ratchet*(`http://goratchet.com/`) cujo *framework* é baseado em *Bootstrap*. Para ambas as aplicações foi desenvolvido código Cascading Style Sheets (*CSS*) e código *Javascript*. A biblioteca *JQuery* (`https://jquery.com/`) define-se como essencial para os efeitos produzidos no desenvolvimento do último referido.

Na aplicação *desktop*, a página principal (*Home*) apresenta uma imagem central (Figura 2.1), com algum texto, a convidar o utilizador a testar o serviço. No canto superior direito há hiperligações para *links* que podem interessar, como o do repositório deste projeto, ou o da universidade onde foi desenvolvido, e menção aos desenvolvedores da aplicação. Do lado esquerdo podemos ver um menu vertical com as três páginas principais da aplicação. É através deste que o utilizador poderá navegar entre as diversas páginas da aplicação.

Na página *New music* é possível ao utilizador criar uma nova música. O utilizador pode introduzir o nome da música, a pauta e o registo desejado. Desde

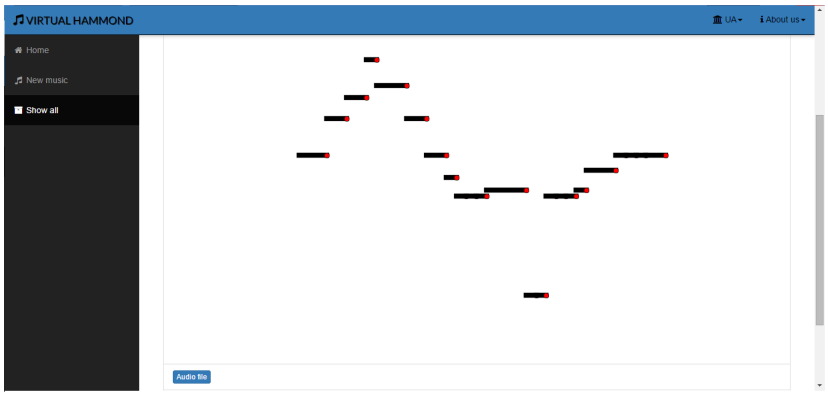

Figura 2.2: Página *Show Info* da aplicação para *desktop*. Podemos ver o gráfico criado pela música "The Simpsons", bem como o botão para ouvir a música.

que a pauta esteja no formato correto *RTTTL* e o registo satisfaça as condições exigidas, a música será criada com sucesso. Caso isso não aconteça, irá ser apresentada uma mensagem de erro. A escolha e aplicação de efeitos é uma decisão do utilizador. Isto irá criar automaticamente uma nova música com uma versão, que será guardada na base de dados, através do servidor.

Na página *Show all* é feita uma listagem de todas as músicas existentes na base de dados. Para cada música, estão disponíveis três botões. O da direita, *Get notes*, que serve para mostrar na mesma página a pauta dessa música. O botão central, *New version*, levará o utilizador a uma página com um campo para escolher um registo e com os efeitos disponíveis. Depois de inseridos estes dados, será criada uma nova versão da música. Finalmente, o botão *Show versions* listará todas as versões disponíveis daquela música, com indicação do registo e dos seus efeitos. Na página referida, o utilizador poderá carregar no botão *Show info*, irá disponibilizar informação sobre a versão escolhida. Irá disponibilizar um gráfico com as notas da música (Figura 2.2), um botão para ser criado e disponibilizado um ficheiro para audição, destacando-se a seguir o registo, os efeitos aplicados e os votos efetuados até ao momento, sendo possível efetuar votos (positiva ou negativamente) da versão em questão.

A aplicação *mobile* foi desenhada para ser o mais similar possível à versão *desktop*. Para se adaptar melhor aos *smartphones* atuais, optámos por colocar um menu inferior horizontal, com os mesmos botões que na versão *desktop* (embora a redirecionar para links diferentes). A página *New music* tem as mesma funcionalidades, sendo bastante similar também na sua aparência em relação à *New music* da versão *desktop*. Também a página *Show all* (Figura 2.3) não difere em quase nada da versão *desktop*, assim como as suas sucessivas ramificações através de botões existentes. A principal diferença é na maneira como as páginas fazem as transições entre si. Usando a funcionalidade *push* disponibilizada pelo *Ratchet*, as páginas não comutam entre si de maneira usual. Só pode haver troca

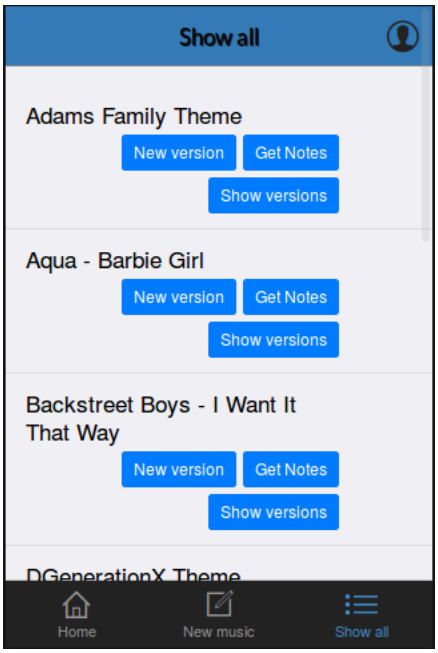

Figura 2.3: Possível página *Show all* da aplicação para *mobile* (ecrã de tamanho 320x480).

de página se houver um evento *touch* nos botões disponibilizados. Quando isso acontece, apenas o conteúdo é alterado entre as páginas. A sua interação em computadores é apenas possível quando são ativados os *touch events* através das opções de cada *browser*.

2.2 Aplicação Principal

A aplicação principal é o módulo responsável por interligar o ambiente apresentado ao utilizador pela interface ao núcleo de processamento e armazenamento de informação, como a base de dados e os módulos de interpretação e sintetização. A sua estrutura pode ser subdividida em três partes: módulos importados, funções de serviços *Cherrypy* e configuração do servidor.

Os módulos utilizados nesta aplicação são orientados para as necessidades e especificidades do código produzido e apresentam as seguintes funções:

- *sqlite3*: disponibiliza uma interface entre a aplicação desenvolvida em linguagem Python e a base de dados relacional do tipo *SQLite*;
- *os* e *os.path*: permite a manipulação de ficheiros e diretórios no ambiente Python;
- *cherrypy*: permite o desenvolvimento de aplicações Web, apresentando uma

interface de disponibilização de um servidor Hypertext Transfer Protocol (*HTTP*);

- *sys*: garante o acesso a variáveis de sistema e de do interpretador;
- *json*: usado na manipulação de objetos do tipo *JSON*;
- *matplotlib.pyplot* as plt: biblioteca de criação de gráfico 2D (pode não funcionar corretamente quando utilizada em sistemas *Windows*);
- *socket*: módulo que disponibiliza o acesso ao sockets de comunicação usados para a comunicação entre aplicações;
- *urllib2*: permite a manipulação e ligação a URLs e redirecionamento de informação.

Para além destas bibliotecas existentes para o desenvolvimento em *Python* estão definidas outras que foram implementadas especificamente para o projeto em questão:

- *from inter import interp*: usa a função *interp* do módulo *inter*, cuja função é a interpretação das pautas em formato RTTL:
- *from graph_creator import createForm*: usa a função *createForm* do módulo *graph_creator* , responsável pela criação do gráfico de frequências das notas;
- *from sint import sintetizador*: usa a função *sintetizador* do módulo *sint*, que permite formar a onda de cada música.

A configuração do servidor *Cherrypy* assenta na definição dos diferentes recursos que podem ser invocados pela aplicação. São configurados aspetos como a localização de ficheiros, endereços de acesso e definição da estrutura de caminhos da aplicação.

O acesso à aplicação está configurado de forma a que possa ser efetuado por qualquer dispositivo compatível através da rede. Num processo transparente para o utilizador e independente da sua localização a função *get_ip_address* estabelece uma ligação ao domínio *google.com*, que tem como único fim a possibilidade de obter o endereço IP da máquina que nesse momento disponibiliza a aplicação.

```
1 def get_ip_address():
2
3     s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
4     s.connect(('google.com', 0))
5     ipaddr=s.getsockname()[0]
6     return ipaddr
```

Assim, o acesso pode ser feito através da rede, permitindo que os dispositivos, sejam eles fixos ou móveis possam interagir com a aplicação. No momento em que a aplicação é ativada, é apresentada a informação de acesso à aplicação, com a indicação dos passos a seguir para utilização por parte de dispositivos móveis.

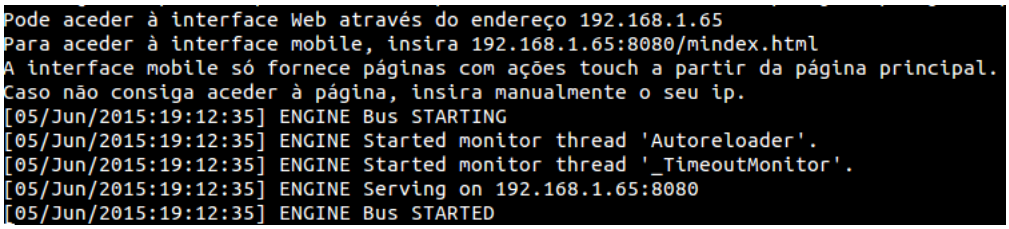

Figura 2.4: Mensagem apresentada aquando da iniciação da aplicação

Os recursos que a aplicação usa e disponibiliza são muito importantes e necessitam de uma indicação clara em relação ao caminho para chegar até eles. O primeiro passo é o estabelecimento de uma estrutura baseada na localização absoluto da aplicação na máquina que a corre. Todos os recursos são chamados e disponibilizados com base nesta localização e num caminho relativo definido a partir daí.

```
1  current_dir = os.path.dirname(os.path.abspath(__file__))
```

A informação guardada pela variável *current_dir* permite a disponibilização de recursos como ficheiros *CSS* ou *JavaScript*, imagens, páginas HyperText Markup Language (*HTML*) e outros recursos através de um mapeamento estático do seu caminho. Como é possível verificar o caminho para o recurso é mapeado a partir da raiz definida através da variável *current_dir* e seguindo a orientação da estrutura indicada para cada caso.

```
1  conf = {'/': {'tools.staticdir.root': current_dir},
2          '/css': { 'tools.staticdir.on': True,
3                    'tools.staticdir.dir': 'virtualhammond/css'},
4          '/js': {  'tools.staticdir.on': True,
5                    'tools.staticdir.dir': 'virtualhammond/js'},
```

Em termos estruturais, a parte do processamento assenta em várias funções que disponibilizam serviços utilizados pela interface Web. A classe *Root* engloba as funções *index*, *createSong*, *createInterpretation*, *listSongs*, *listSongFiles*, *getNotes*, *updateVotes*, *getWaveFile* e *getWaveForm*.

As funções *createSong* e *createInterpretation*, são responsáveis pela criação de novas músicas ou de interpretações. Em termos de processamento apenas definem novas entradas na base de dados. A função *listSongs* tem como finalidade a apresentação de dados sobre as diferentes músicas. As restantes funções são mais especificas, uma vez que utilizam parâmetros que identificam certos elementos. A função *getNotes* disponibiliza a pauta de uma música, enquanto que a *listSongFiles* permite acesso a vários dados de interpretações especificas, tal como efeitos e registos definidos, nome e número de votos. As funções *getWaveFile* e *getWaveForm* são responsáveis pelo processamento da criação dos ficheiros de som e imagem da interpretação. Como cada versão permite a atribuição de votos negativos e positivos a comunicação do número de votos em cada atualização é feita pela função *updateVotes* que guarda a informação dos votos na base de dados.

2.3 Interpretador de Pautas

O interpretador de pautas (ficheiro "inter.py") irá receber uma *String* (que representará a pauta) no formato *RTTTL*. Se não estiver no formato correto (se tiver espaços, se existirem mais ou menos do que dois ":", se houver um oitava sem nota, se houver uma nota ou uma oitava inexistentes, ...) lançará erro e retornará uma lista vazia. Foi feito um teste (usando *pytest*) ao módulo, para garantir que assegurava o controlo dos erros, e que lançava as pautas corretas.

Caso esteja no formato correto, deverá armazenar as oitavas de referência, o pulso de referência e o andamento. Para cada nota, deverá fazer as contas à sua duração e à sua frequência (de acordo com uma *Lookup Table* com todas as frequências possíveis de notas) e adicionar a uma lista estes pares. Cada nota deverá ter obrigatoriamente o seu tom. Poderá ter opcionalmente, antes da nota, o seu valor (se não tiver, presumir-se-há valor igual ao pulso de referência), ou depois da nota, um ponto (indica duração aumentada em 50%) ou uma oitava (de 0 a 8, se não existir presume-se igual à oitava de referência). No final da pauta, essa lista será retornada.

2.4 Sintetizador

O sintetizador é responsável por formar a onda de cada música dado um conjunto de frequências e um certo registo. Recebendo uma lista de pares (duração, frequência) enviada pelo interpretador e um registo com nove algarismos, de 0 a 8 (verificado pelo *JavaScript* existente), o sintetizador irá somar nove ondas com um certo múltiplo da frequência (1/2, 3/2, 1, 2, 3, 4, 5, 6, 8), ponderado pelo algarismo do registo nesse dígito, multiplicado pela amplitude e pela onda formada. Assim, é possível formar uma onda "analógica"com uma certa frequência de amostragem (44100 Hz) através de valores discretos.

A fórmula para este cálculo é

$$\sum_{j=1}^{9} amplitude * \frac{reg[j]}{8} * \sin(\frac{2 * \Pi * f * mul[j] * i}{rate}) \tag{2.1}$$

onde *reg* é uma lista com os nove dígitos presentes no registo, f é a frequência, *mul* é uma lista com os múltiplos da frequência (descritos acima), i é um valor crescente que irá indicar o tempo e *rate* será a frequência de amostragem, neste caso, 44100 Hz.

Após fazer a soma das ondas, a onda final terá que ser normalizada para depois poder ser enviada para o processador de efeitos para ser construído um ficheiro de formato WAVEform audio file format (*WAVE*), que o sintetizador irá retornar o seu diretório ao servidor.

2.5 Processador de Efeitos

O processador (ficheiro "effects_processor.py") deverá receber uma lista de tons pares (frequência, *samples*) vindos do sintetizador, onde *samples* são as amostras que possui a onda criada para essa frequência. O sintetizador irá criar um ficheiro no formato *".wav"* e irá aplicar os seguintes efeitos, caso sejam pedidos pelo sintetizador:

- **Chorus**: Consiste em introduzir à atual onda outra onda de frequência ligeiramente superior (neste caso, 30 Hz), para cada frequência.
- **Percussion**: Soma uma onda à atual com um múltiplo da frequência (neste caso, 4 vezes a frequência atual), com uma amplitude que decai ao longo do tempo, até se anular no instante final.
- **Tremolo**: Faz variar ligeiramente a amplitude do sinal. Ao sinal atual, soma-o com ele próprio multiplicado com uma onda de amplitude mínima (neste caso, 0.3).
- **Distortion**: Eleva cada tom da nota a um número natural. Por ser bastante inconfortável, o aplicado no processador de efeitos foi um efeito semelhante, mas mais eficaz (efeito original encontra-se comentado no código, cujos comentários podem ser retirados para testes). O efeito aplicado foi multiplicar o tom atual por ele próprio, multiplicado depois por uma amplitude mínima (0.01).
- **Echo**: Aplica eco a uma música, somando um sinal futuro ao sinal atual, com uma atenuação na amplitude (no código tem amplitude 0.2).
- **Envelope**: Deverá modelar cada nota de acordo com as características do instrumento, simulando assim mais fielmente a sua reprodução. Para frequências iguais seguidas, o efeito deverá ser anulado e a amplitude das notas deverá ser igual.

Para cada efeito, depois de aplicado, serão aplicados um de dois efeitos: o *normalize*, que normalize as ondas com a amplitude máxima e mínima e evita *clipping*, ou o efeito *clipping*, que não o evita.

Capítulo 3

Implementação

3.1 Interface Web

A interface *Web*, tal como mencionado no capítulo que apresenta a estrutura, foi implementada para uma versão *desktop* e *mobile*. Os ficheiros *.html* das mesmas podem ser facilmente identificados nas pastas *virtualhammond* no diretório principal. A versão *desktop* é composta por sete ficheiros do tipo *HTML*, nomeadamente: *developers.html, index.html, newmusic.html, newversion.html, showall.html, showinfo.html e showversions.html*. Cada um destes ficheiros representa uma página com um propósito específico de interação com o utilizador. Os ficheiros respetivos à implementação *mobile* começam pela letra *m* seguidos pelo nome igual ao do ficheiro análogo para a implementação *desktop*, à excepção do *developers* que se apresenta sob um *modal* disponível em cada página *mobile*.

Análise do código base constituinte

Vejamos a implementação da página *index.html* e serão comentadas as seguintes páginas apenas nas suas variações.

O *head* da página *index.html* é igual ao das restantes páginas. Aqui se apresenta o seu código:

```
1  <head>
2      <meta charset="utf-8">
3      <meta http-equiv="X-UA-Compatible" content="IE=edge">
4      <meta name="viewport" content="width=device-width, initial-scale
       =1">
5      <meta name="description" content="Play your music here!">
6      <meta name="keywords" content="Web, App, Virtual Hammond, Music,
       Universidade de Aveiro, Computing">
7      <meta name="author" content="Abel Fernandes Neto, Daniel Azevedo
       Alves, Diogo Daniel Soares Ferreira, Luís Davide Jesus Leira">
8
9      <title>Virtual Hammond</title>
10
```

```
11      <!-- Bootstrap Core CSS -->
12      <link href="css/bootstrap.min.css" rel="stylesheet">
13
14      <!-- Custom CSS -->
15      <link href="css/sb-admin.css" rel="stylesheet">
16
17      <!-- Custom Fonts -->
18      <link href="font-awesome/css/font-awesome.min.css" rel="
        stylesheet" type="text/css">
19
20      <!-- Styles CSS-->
21      <link rel="stylesheet" href="css/styles.css">
22
23      <!-- Virtual Hammond font -->
24      <link href='http://fonts.googleapis.com/css?family=Lato' rel='
        stylesheet' type='text/css'>
```

Tal como se pode observar, as primeiras linhas destinam-se a definir os dados, *meta*, que poderão ser usados como por exemplo por motores de busca. Define-se também o título e seguidamente os *links* para os ficheiros *CSS* e para a fonte da letra do título da página. Tal como mencionado no capítulo anterior foi usado um *template Bootstrap* (nomeadamente, *SB Admin* `http://startbootstrap.com/template-overviews/sb-admin/`).

Como forma de definir o *layout* da página todo o conteúdo da mesma é editado dentro da *div* com o *id="wrapper"*. Dentro deste bloco a página é definida em dois blocos ao mesmo nível. O bloco de navegação:

```
1       <nav class="navbar navbar-inverse navbar-fixed-top" role="
        navigation">
```

E o bloco com o conteúdo da página *index.html*:

```
1       <div id="page-wrapper">
```

É de notar que o que muda em todas as páginas *HTML* (que pertencem à aplicação *desktop*) é precisamente o conteúdo deste último bloco a ser mencionado, isto visto o esquema de navegação ser igual em todas.

A página *mindex.html*, que se apresenta como página inicial da aplicação *mobile*, contém um *head* bastante semelhante ao referido anteriormente, diferindo em pequenos pormenores como os ficheiros *CSS* e ficheiros que contém os *Scripts* (ou seja, código *JavaScript*). Em relação à aplicação *desktop*, os *Scripts* encontram-se no fim de cada página.

Na página *New music* o conteúdo é apresentado num formato de introdução de dados que compõem uma música e um botão que confirma este formulário preenchido devidamente pelo utilizador. Baseia-se no seguinte código:

```
1               <div class="form-group" id="newmusic">
2                   <label>Create a new song here!</label>
3                   <br>
4                   <input class="form-control" type="text"  required
        placeholder="Song name" id="nameinput">
5                   <br>
```

```
6                   <input class="form-control" type="text" required
    placeholder="Enter the song in RTTTL format" id="songinput">
7                   <br>
8                   <input class="form-control" type="number"
    required placeholder="Song register (9 numbers between 0 and 8)"
    id="registerinput">
9                   <br>
10                  <div class="checkbox">
11                      <label><input type="checkbox" value="" id="
    echoinput">Echo</label>
12                      <label><input type="checkbox" value="" id="
    treminput">Tremolo</label>
13                      <label><input type="checkbox" value="" id="
    distinput">Distortion</label>
14                      <label><input type="checkbox" value="" id="
    percinput">Percussion</label>
15                      <label><input type="checkbox" value="" id="
    chorusinput">Chorus</label>
16                      <label><input type="checkbox" value="" id="
    envinput">Envelope</label>
17                  </div>
18                  <br>
19                  <button type="submit" class="btn btn-success" id=
    "addsong">Add song</button>
20              </div>
```

Este código encontra-se escrito da mesma forma na aplicação *mobile*, à excepção das classes atribuídas. No entanto, em termos de aparência para o utilizador é praticamente igual. Em ambos os casos, é utilizado *JavaScript* para verificar se os dados preenchidos se encontram num formato correto, transferindo posteriormente cada campo para o servidor, o qual se encarrega de assegurar que é guardado na base de dados da aplicação. Na página *newversion.html* o conteúdo é praticamente o mesmo, à excepção dos id's e das entradas que pedem o nome e a pauta da música (linhas 4 e 6 do código anterior), visto que já não são necessárias aquando da criação de uma nova versão.

Nas páginas *showall.html* e *showversions.html* o seu conteúdo encontra-se a cargo do *JavaScript*, sendo que este recebe as músicas ou versões (dependendo do caso) e trata de as listar ordenadamente com as opções já referidas no capítulo referente à estrutura da interface *Web*. O mesmo acontece com as páginas correspondentes na aplicação *mobile*.

A página *showinfo.html* apresenta o seu conteúdo com suporte no código *JavaScript*, tendo como objetivo disponibilizar as informações relativas a uma versão específica. Irá ser listada a seguinte informação: o nome da música e versão, um gráfico das notas da música, um botão que permite gerar e reproduzir o seu ficheiro áudio através de um *player*, o registo, os efeitos aplicados, o número de votos positivos e negativos atribuídos e a possibilidade de votar essa mesma versão através das imagens de *like* e *dislike*. A página correspondente na aplicação *mobile* efetua as mesmas ações, diferindo apenas no pormenor do *loading* implementado, que se apresenta mais simples e igualmente eficaz para o que se pretende.

Por último, a página *developers.html* disponibiliza ao utilizador informações como os desenvolvedores da aplicação, a possibilidade de contactar cada um por e-mail e um pequeno texto informativo. Na aplicação *mobile*, não existe uma página *HTML* que faça correspondência direta com esta, sendo apresentada através de um *modal* disponível em qualquer página *HTML* desta versão mais reduzida e simplista, contendo apenas informações básicas sobre os criadores e uma referência ao repositório onde se encontra o desenvolvimento deste projeto.

Este módulo teve como principal contribuidor Luís Leira, que desenvolveu aproximadamente 62,5% do módulo e contou com o apoio do Daniel Alves, desenvolvendo cerca de 32,5%, sendo a restante percentagem dividida igualmente pelo Abel Neto e pelo Diogo Ferreira.

3.2 Aplicação Principal

A aplicação principal apresenta-se como um ponto de conexão entre os diferentes serviços. Sempre que o utilizador interage com o interface, despoleta uma funcionalidade da aplicação. Todo o código implementado neste módulo assenta na linguagem *Python*, sendo evocadas várias bibliotecas para a execução das tarefas previstas. As funções desenvolvidas para responder aos pedidos da interface Web estão definidas dentro da classe *Root*, que foi fixada na configuração como a raiz da estrutura.

Este módulo teve como principal contribuidor Daniel Alves, que desenvolveu aproximadamente 70% do módulo e contou com a ajuda de Diogo Ferreira e Luís Leira numa contribuição de partes iguais.

3.2.1 A implementação da base de dados e a função *index*

Neste projeto o armazenamento de informação é essencial pelo que foi necessário definir uma estrutura capaz de lidar com pedidos de forma eficaz. A flexibilidade que a linguagem Python tem facilita a interligação com a linguagem Structured Query Language (*SQL*). Desta forma é possível obter um programa organizado, fluído e coerente. A base de dados desenvolvida para esta aplicação assenta no módulo *sqlite3* e foi construída com a estrutura apresentada na figura. Existem duas tabelas chamadas *musics* e *interpretations*. A tabela *musics* guarda informação relativa às diferentes músicas que são inseridas na aplicação. Os três campos que constituem esta tabela são *music_id*, que está definido como *INTEGER PRIMARY KEY AUTOINCREMENT*, *name* e *stave* definidas como *TEXT*. A tabela *interpretations* guarda a informação das diferentes versões criadas para uma música. As versões caracterizam-se por poderem ser criadas com base em vários atributos diferentes. Para uma dada interpretação é possível definir um registo e combinar até seis efeitos diferentes. A tabela apresenta como chave primária o atributo *interp_id*, que está definido como *INTEGER PRIMARY KEY AUTOINCREMENT*. Existem ainda campos para *register* (*INTEGER*), *effect_echo* (*TEXT*), *effect_tremolo* (*TEXT*), *effect_perc* (*TEXT*), *ef-*

fect_chorus (*TEXT*), *effect_dist* (*TEXT*) e *effect_textttenv* (*TEXT*). A informação das votações é guardada nas colunas *posvotes* e *negvotes* na forma de *INTEGER DEFAULT* 0. Desta forma de cada vez que uma nova interpretação é criada este campo é preenchido de forma automática com o valor 0. A conexão entre as duas tabelas é assegurada pela definição de uma chave estrangeira *music_id* na tabela interpretations (Figura 3.1). Desta forma é possivel obter dados como o nome ou a pauta de uma música a partir do *interp_id*.

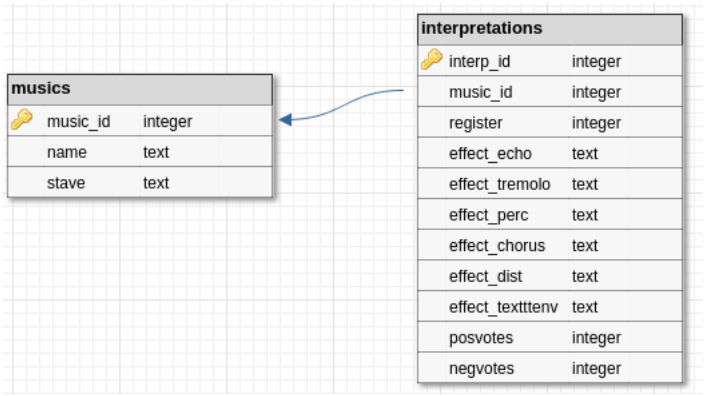

Figura 3.1: Tabelas da base de dados

O comando *CREATE TABLE IF NOT EXISTS* garante a criação da base de dados e respetivas tabelas apenas no caso de estas não existirem.

```
con = sqlite3.connect('music.db')
cur = con.cursor()
reload(sys)
sys.setdefaultencoding('utf8')
#cria tabela com informaćao de nome e pauta da música (se nao existir)
cur.execute('''CREATE TABLE IF NOT EXISTS musics (music_id INTEGER PRIMARY KEY AUTOINCREMENT, name TEXT, stave TEXT)''')
#cria tabela com informaćao das diferentes interpretaćoes e respetivos registos, e efeitos (se nao existir)
cur.execute('''CREATE TABLE IF NOT EXISTS interpretations (interp_id INTEGER PRIMARY KEY AUTOINCREMENT, music_id INTEGER,
  register INTEGER, effect_echo TEXT, effect_tremolo TEXT,
  effect_perc TEXT, effect_chorus TEXT, effect_dist TEXT,
  effect_textttenv TEXT, posvotes INTEGER DEFAULT 0,
  negvotes INTEGER DEFAULT 0, FOREIGN KEY(music_id) REFERENCES music(music_id))''')
```

Algumas complicações com a codificação em 8-bit Unicode Transformation Format (UTF-8) obrigaram à introdução de um forçar de codificação, que assim garante um funcionamento adequado da aplicação.

No momento inicial de acesso à página o servidor corre o código descrito nesta função. Desta forma a página *index.html* é disponibilizada pela função *index* descrita na classe *Root*.

3.2.2 As funções *createSong* e *createInterpretation*

As funções *createSong* e *createInterpretations* têm como tarefa a criação de novas entradas nas tabelas da base de dados. Apesar do nome sugerir a criação do som da música ou da interpretação, estas funções fazem apenas leituras e atualizações. O acionamento da função *createSong* dá-se sempre que o utilizador optar por criar uma nova música através da página *newmusic.html* (ou *mnewmusic.html* na versão mobile). Do lado da interface o utilizador introduz a informação do nome e pauta da música, que é guardada na tabela musics. Na aplicação criada foi considerado que a criação de uma música implica a criação da primeira versão. Assim é pedido ao utilizador um valor do registo e os efeitos a aplicar. Isto implica que a função *createSong* tenha muitos parâmetros de entrada, no total nove.

```
1  @cherrypy.expose
2  #adiciona uma nova música com base no nome, notas da pautas,
3  def createSong(self, name, notes, register, effect_echo,
     effect_tremolo, effect_perc, effect_chorus, effect_dist,
     effect_textttenv):
```

A interface Web garante a validade dos parâmetros introduzidos, à exceção da pauta. Esta é verificada pelo módulo interpretador de pauta. Caso o resultado da interpretação seja uma lista vazia a pauta não é aceite e a entrada para a música não é criada.

```
1     pauta = interp(notes)
2     if len(pauta) == 0:
3       returnjson =  '{"message": "The music '+ name + ' is not in the
       right format."}'
4     else:
5
6       #inserćao na tabela musics o nome e a pauta introduzida pelo
      utilizador
7       cur.execute('''INSERT INTO musics (name, stave) VALUES (?,?)'''
      ,(name1, notes1))
8       con.commit()
9       cur.execute('''SELECT music_id FROM musics WHERE name LIKE ?'''
      ,(name1,))
10      musicid = cur.fetchall()[0][0]
11
12      # Cria interpretaćao da música
13      url = "http://"+cherrypy.server.socket_host+":8080/
      createInterpretation?musicid="+str(musicid)+"&register="+str(
      register)+"&effect_echo="+effect_echo+"&effect_tremolo="+
      effect_tremolo+"&effect_perc="+effect_perc+"&effect_chorus="+
      effect_chorus+"&effect_dist="+effect_dist+"&effect_textttenv="+
      effect_textttenv+""
14      data = urllib2.urlopen(url).read()
15
16      #criaćao de uma string com sintaxe que permite a codificaćao em
       JSON
17      returnjson =  '{"message": "The music '+ name + ' was
      sucessfully added."}'
```

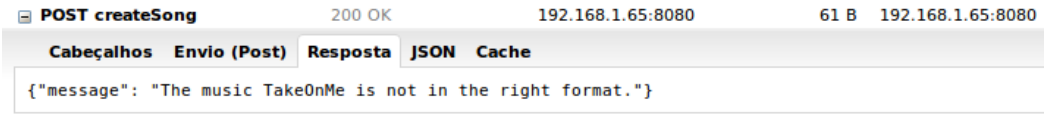

Figura 3.2: Aviso de pauta com formato errado

A resposta nesta situação é uma mensagem que identifica a pauta como não estando no formato correto (Figura 3.2). Se a validação da pauta for positiva, o processo avança com a conexão à base de dados e a inserção dos dados na tabela *musics*. Para garantir a criação da primeira versão é necessário fazer o reencaminhamento para a função *createInterpretation*. Isto é conseguido através do módulo *urllib2*, que lida com o redirecionamento para outros links. Desta forma, os valores do registo e dos efeitos são passados à função *createInterpretation*, que depois insere os dados de uma nova interpretação na tabela *interpretations* da base de dados.

Outro momento em que a função *createInterpretation* é chamada é aquando da criação de uma nova versão a partir de uma música existente na base de dados. Os valores que recebe são introduzidos na tabela *interpretations*, usando o comando *INSERT INTO*. Cada parâmetro recebido é colocado na respetiva coluna, sendo que efeitos que não tenham sido escolhidos, recebem um valor de *NONE*.

Um aspeto comum às funções implementadas é a forma como envia os dados para a interface Web. A comunicação é feita com base no formato *JSON* e estrutura-se de forma a passar os elementos a apresentar (ou utilizar) na página, sendo os cabeçalhos de comunicação identificados com o respetivo formato. Um exemplo é o valor devolvido pela função *createInterpretation*. Neste caso pretende-se dar uma informação de sucesso. O elemento *JSON* é definido com a chave mensagem e o respetivo valor.

```
1    returnjson = '{"mensagem":"The new version was created with
     success!"}'
```

3.2.3 As funções *listSongs* e *listSongFiles*

O papel destas funções é de apresentar dados sobre músicas e interpretações existentes na base de dados. A apresentação de todas as músicas presentes na base de dados é conseguida com a função *listSongs*. O facto de esta não necessitar de parâmetros define-a como uma função de apresentação de valores gerais sobre uma música. A consulta da base de dados é conseguida com recurso ao comando *SELECT*, especificando os atributos a disponibilizar e como devem ser ordenados. A natureza mais complexa desta resposta obriga à criação de um elemento *JSON* que incorpore todos os elementos encontrados. Neste caso, recorre-se a um elemento *JSON* na forma de lista. Isto permite que cada linha da lista diga respeito a uma entrada da base de dados.

A função *listSongFiles* tem uma incumbência mais especifica, uma vez que

trata a informação referente a uma música. O parâmetro que recebe (*musicid*) é usado na pesquisa dos dados e permite a obtenção dos valores relativos às diferentes interpretações existentes. A garantia de uma resposta não vazia é conseguida pela imposição de um elemento condicional. Caso este não se encontre vazio é devolvido no formato *JSON*, recorrendo-se ao mesmo processo de codificação da *string* concatenada com a informação pesquisada.

```
1     if len(songlist) == 0:
2       returnjson = '{"message": "unsucess"}'
3     #criaćao de uma string com sintaxe que permite a codificaćao em
      JSON
4     else:
5       returnjson =  '['
6       for x in songlist:
7         returnjson+= '{"interpid":'+str(x[0])+',"nome":"'+x[1]+'","
      register":"'+str(x[2])+'","effect_echo":"'+unicode(x[3])+'","
      effect_tremolo":"'+unicode(x[4])+'","effect_perc":"'+unicode(x[5])
      +'","effect_chorus":"'+unicode(x[6])+'","effect_dist":"'+unicode(x
      [7])+'","effect_texttt":"'+unicode(x[8])+'","posvotes":"'+str(x
      [9])+'","negvotes":"'+str(x[10])+'"},'
8       returnjson=returnjson[:-1]
9       returnjson+=']'
```

Na Figura 3.3 é possível ver parte da resposta enviada pela aplicação para interface Web na função *listSongFiles*.

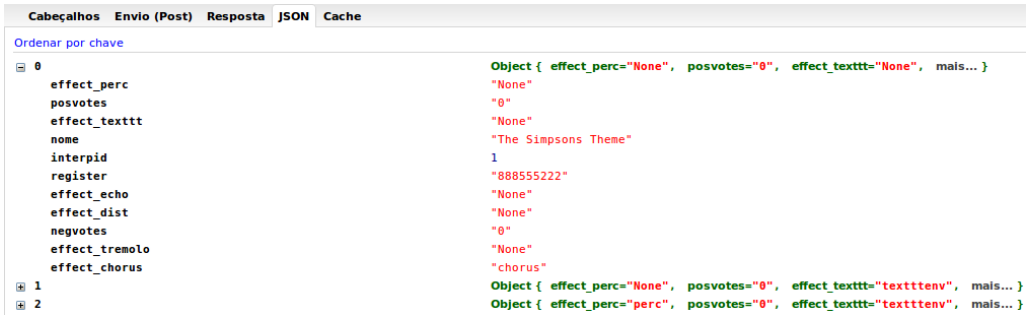

Figura 3.3: Informação enviada para a Interface Web

3.2.4 As funções *getNotes* e *updateVotes*

A interação dos utilizadores com a aplicação vai para além da criação de músicas e respetivas interpretações. De forma a criar uma ligação maior às interpretações que criam, os utilizadores podem atribuir votos positivos ou negativos. A aplicação principal intervém neste processo apenas como módulo de armazenamento dos votos na base de dados. A função *updateVotes* é acionada cada vez que os botões *likes* ou *dislikes* são premidos, recebendo a informação do respetivo número de votos. A escolha do botão implica a colocação da variável "contrária"

a zero. Desta forma a função analisa os parâmetros *neg* e *pos* e decide qual necessita de ser atualizado. Na base de dados é possível definir um novo valor com o comando *UPDATE* em conjunto com o comando *SET*.

```
1     if neg == "0" :
2       cur.execute('''UPDATE interpretations SET posvotes = ? WHERE
      interp_id LIKE ?''', (pos,interpretationid,))
3     elif pos == "0" :
4       cur.execute('''UPDATE interpretations SET negvotes = ? WHERE
      interp_id LIKE ?''', (neg,interpretationid,))
5     con.commit()
```

A função *getNotes* tem como tarefa a pesquisa e envio da pauta relativa a uma música específica. O parâmetro *musicid* que recebe permite a procura da respetiva pauta na tabela *musics* da base de dados.

3.2.5 As funções *getWaveForm* e *getWaveFile*

Todas as funções mencionadas até ao momento têm como tarefa a pesquisa, inserção ou atualização da base de dados, não sendo produzidos ficheiros com os quais o utilizador pode interagir ou observar. As funções *getWaveForm* e *getWaveFile* são responsáveis por implementarem tarefas e chamarem módulos de criação de ficheiros de som e gráficos de notas.

A função *getWaveForm* implementa a criação do gráfico definido por ramos das notas de uma música. Em termos de execução, a função usa o parâmetro recebido para pesquisar pela pauta na tabela *musics* da base de dados. A pauta é interpretada pelo módulo interpretador de pauta e passado ao módulo de criação de gráficos. A função não devolve o ficheiro imagem, mas apenas o nome do ficheiro para a interface Web.

```
1   def getWaveForm(self, musicid):
2     con = sqlite3.connect('music.db')
3     cur = con.cursor()
4     #pesquisa pela pauta de uma música associada a uma interpretaćao
      especifica
5     cur.execute('''SELECT stave FROM musics WHERE music_id LIKE ?''',
       (musicid,))
6
7     #pauta guardada na variável notes
8     stave=cur.fetchall()[0][0]
9     #chamado o interpretador de stave
10    notes = interp(stave)
11
12    createForm(notes)
13    #pauta da música depois de interpretada
14
15
16    returnjson =  '{"message":"'+'notes.png'+'"}'
17    #codificaćao da variavel no formato JSON
18    returnmessage = json.loads(returnjson)
19
```

```
20      #definićao do tipo de cabećalho para a comunicaćao HTTP
21      cherrypy.response.headers["Content-Type"] = "application/json"
22
23      #devolućao no formato JSON para comunicaćao com o Javascript
24      return json.dumps(returnmessage)
```

A função *getWaveFile* realiza a tarefa de gestão e passagem de valores aos vários elementos que estão envolvidos na criação do ficheiro de som. O processo começa pela pesquisa na base de dados da informação relativa à pauta, efeitos e registo de uma certa interpretação. A forma como os efeitos se encontram armazenados na base de dados obriga a algum processamento antes de poder enviar os dados a outros módulos. A tabela *interpretations* tem uma coluna para cada efeito, sendo este preenchido pelo nome do efeito, caso tenha sido selecionado, ou com *NONE*, caso não tenha sido escolhido. Esta informação necessita de ser inserida numa lista, pelo que passa por um processo de validação e definição dos efeitos escolhidos.

```
1       # Adiciona os efeitos a uma lista para enviar para processador de
         efeitos
2       effects = []
3       if not(effect_echo == "None"):
4         effects.append("echo")
5       if not(effect_tremolo == "None"):
6         effects.append("tremolo")
7       if not(effect_chorus == "None"):
8         effects.append("chorus")
9       if not(effect_dist == "None"):
10        effects.append("dist")
11      if not(effect_perc == "None"):
12        effects.append("perc")
13      if not(effect_textttenv == "None"):
14        effects.append("textttenv")
```

Tal como na função *getWaveForm* a pauta é enviada ao interpretador de efeitos que devolve uma lista de tuplos. O processo de criação do ficheiro no formato *WAVE* é despoletado pela chamada do módulo sintetizador, que recebe o valor do registo, a lista interpretada e a lista de efeitos. O resultado deste processo é um ficheiro de som, cujo caminho relativo é enviado para a interface Web.

3.2.6 Criador de gráfico

O módulo de criação do gráfico (ficheiro "graph_creator.py") irá receber a lista produzida pelo interpretador de pauta. Esta lista é constituída por pares (frequência, tempo), que no gráfico são representadas por linhas horizontais. Na manipulação e criação do gráfico é usado o módulo *matplotlib*. Por forma a definir uma janela de visualização adequada para os dados de cada música procura-se os valores mínimos e máximos das frequências presentes na lista. O gráfico é constituído por linhas horizontais que unem dois pontos com duas coordenadas. Assim o ponto inicial corresponde ao par (*xinitial*, *freq*) e o ponto final ao par

(*xfinal*, *freq*). Para o efeito vermelho na extremidade de cada ramo é sobreposto um segundo gráfico constituído apenas pelos pontos (*xfinal*, *freq*). Retirando os eixos e configurando a janela de visualização vertical com a ajuda dos valores mínimos e máximos das frequências é possível desenhar um gráfico como o da Figura 3.4. O gráfico é guardado localmente sendo possível visualizá-lo através

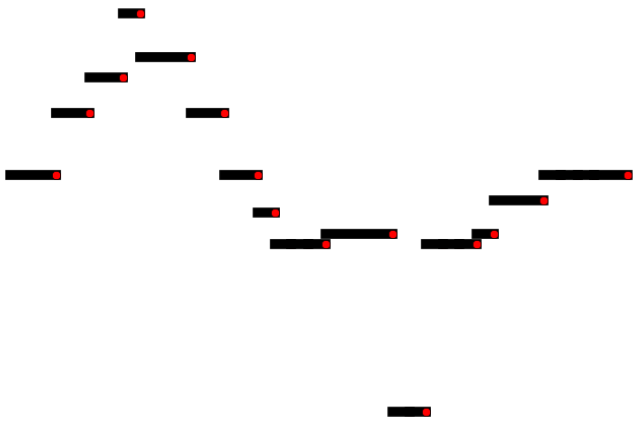

Figura 3.4: Gráfico das frequências das notas

da interface Web. É importante referir que cada vez que o gráfico é criado e guardado é necessário limpar a variável *plt*, para que se possa gerar um novo gráfico. A não existência desta condição leva à apresentação do mesmo gráfico para qualquer música.

```
1  for (time,freq) in data:
2    #Procura o menor e o maior valor das frequencias
3    if freq < freqmin:
4      freqmin = freq
5    if freq > freqmax:
6      freqmax = freq
7
8    #A variavel xfinal representa o valor da abcissa onde termina a
     linha horizontal e resulta da
9    #soma abcissa anterior com o valor do tempo para o qual se mantém
      a frequencia
10   xfinal = xinitial + time
11
12   #criaćao das linhas do gráfico com indicaćao dos pontos e da cor,
      padrao e grossura das linhas
13   plt.plot((xinitial, xfinal), (freq,freq), 'k-', linewidth=7)
14
15   #criaćao de um segundo gráfico de pontos nas extremidades finais
     de cada linha
16   plt.plot(xfinal,freq, 'ro')
17   xinitial = xfinal
18
```

```
19  #O intervalo de visualizaćao para o eixo vertical vai de uma valor
      50 unidades inferior ao menor valor
20  #de frequencias até um valor 50 unidades superior ao maior valor de
      frequencias
21  plt.ylim(freqmin-50, freqmax+50)
22  plt.axis('off')
23  if os.path.isfile('virtualhammond/images/notes.png'):
24    os.remove('virtualhammond/images/notes.png')
25  plt.savefig('virtualhammond/images/notes.png', dpi=100)
26  plt.clf()
```

3.3 Interpretador de Pautas

O interpretador, após receber uma pauta em formato *String*, começa por verificar se a pauta recebida tem dois ":". Se não tiver, lançará erro. Depois, tentará encontrar, após o primeiro ":", os valores de 'd', 'o' e 'b'. Caso haja algum erro no formato (espaço a mais, vírgula no local errado, valores inválidos), lançará erro. Caso não existam 'b', 'o' ou 'b', serão definidos os valores por defeito (d=4, o=5, b=63).

Após definido o formato inicial da pauta, tentará ler após o último ":". Assim, entrará numa iteração que, enquanto houver notas, fará o mesmo para todas. Primeiro, tenta ler o valor da nota. Caso o valor seja inválida, lançará uma erro. Caso esse valor não exista, será definido como valor o valor por defeito. Depois, tentará ler o tom da nota. Este elemento é o único que é obrigatório em cada nota, caso não exista, o programa lança erro. Quando deteta a nota, deteta também se é para aumentar meio-tom nessa nota, pelo sinal "#"ou "_".

Após ter a nota guardada, irá detetar a existência de ponto, que indicará se a duração dessa nota será aumentada em metade. Finalmente, irá detetar a oitava existente em cada nota. Caso seja inválida, lança um erro. Caso não exista, a oitava será a definida no início da pauta.

Para cada nota, será calculada uma duração ((4/(valor da nota)*(60/'b')). Se houver ponto, a duração será aumentada em metade. Também será calculada a frequência para cada nota, baseado numa *Lookup Table*. Finalmente, os dois valores serão adicionados em forma (duração, frequência) a uma lista que será devolvida. Esse excerto de código está exposto abaixo:

```
1
2       # Calcula a duraćao da nota e guarda-a numa lista
3       dur = (4/float(v)) * (60/float(b))
4       if dot:
5         dur = 1.5*dur
6
7       dura.append(float(dur))
8
9       # Calcula a frequencia fundamental da nota e guarda-a numa
      lista
10      # Calcula as notas baixando duas oitavas da original.
11
```

```
12          if t == "p":
13            freq = 0
14          else:
15            freq = lut[t+str(oit-2)]
16
17          frequ.append(float(freq))
18
19
20      # Se tudo correr sem qualquer erro, retorna a pauta.
21      i=0
22      while i<len(dura):
23        pauta.append((dura[i], frequ[i]))
24        i+=1
```

Este módulo teve como principal contribuidor Diogo Ferreira, que desenvolveu o módulo a 100%.

3.4 Sintetizador

O ficheiro *sintetizador.py* vai definir uma função que tem como responsabilidade receber três registos, *notes* que define as notas, *register* que contém nove algarismos de zero a oito, *effects* que define a lista de efeitos, todos eles definidos no *main_app.py*. Assim a função é definida como:

```
1 def sintetizador(data, reg, effects):
```

De seguida é verificado se os dados como *reg* vêm no formato certo pelo seguinte código:

```
1       if len(str(reg))!=9:
2           raise Exception
3       for i in str(reg):
4           if int(i)<0 or int(i)>8:
5               raise Exception
```

Com os dados disponíveis resta apenas implementar simplesmente a fórmula 2.1. O código implementado não é mais do que o preenchimento de um array de *samples*.

Vejamos:

```
1           frame = []
2           info = {}
3           duration = j[0]
4           freq = j[1]
5           i=1
6           while i < rate*duration:
7               frame.append(amplitude*(int(str(reg)[0])/8.0)*math.sin
    (2.0*math.pi*(freq*mult[0])*i/rate)+
8                            amplitude*(int(str(reg)[1])/8.0)*math.sin
    (2.0*math.pi*(freq*mult[1])*i/rate)+
9                            amplitude*(int(str(reg)[2])/8.0)*math.sin
    (2.0*math.pi*(freq*mult[2])*i/rate)+
```

```
10                          amplitude*(int(str(reg)[3])/8.0)*math.sin
    (2.0*math.pi*(freq*mult[3])*i/rate)+
11                          amplitude*(int(str(reg)[4])/8.0)*math.sin
    (2.0*math.pi*(freq*mult[4])*i/rate)+
12                          amplitude*(int(str(reg)[5])/8.0)*math.sin
    (2.0*math.pi*(freq*mult[5])*i/rate)+
13                          amplitude*(int(str(reg)[6])/8.0)*math.sin
    (2.0*math.pi*(freq*mult[6])*i/rate)+
14                          amplitude*(int(str(reg)[7])/8.0)*math.sin
    (2.0*math.pi*(freq*mult[7])*i/rate)+
15                          amplitude*(int(str(reg)[8])/8.0)*math.sin
    (2.0*math.pi*(freq*mult[8])*i/rate))
16              i+=1
17          frame = normalize(frame)
18          info['freq'] = j[1]
19          info['samples'] = frame
20          enddata.append(info)
```

De notar também que na linha 18 do código acima podemos ver que cada *sample* é enviada para a função *normalize* que tem como objetivo limpar o possível *clipping* da onda.

```
1  def normalize(data):
2      max = 0
3      for i in data:
4          if abs(float(i))>max:
5              max = abs(float(i))
6      data2 = []
7      if max==0:     # Para garantir a divisao por número diferente de
       0.
8          return data
9      for i in data:
10         data2.append(float(i)*(32767/float(max)))    #float(i) *
       constante de normalizacao
11     return data2
```

O valor máximo de amplitude suportado é 32767 devido ao número de bits disponíveis para cada *sample*. Assim, a função *normalize* procura pelo valor máximo da função e garante através da linha 10 do código acima que essa *sample* assuma o valor 32767 e normaliza os restantes valores multiplicando-os pela constante de normalização que será o valor máximo permitido pelo sistema a dividir pela *sample* de valor máximo.

Este módulo teve como principal contribuidor Abel Neto, que desenvolveu aproximadamente 80% deste módulo, contando com a ajuda de Diogo Ferreira que obteve assim a restante percentagem.

3.5 Processador de Efeitos

O processador de efeitos começa por criar um ficheiro "song.wav"que será onde guardará a música criada. Será um diretório temporário, pois todas as músicas serão guardadas com o mesmo diretório, sobrepondo-se umas às outras, de modo

a não sobrecarregar a aplicação devido ao tamanho ocupado por cada ficheiro no formato *WAVE*.

Serão explicados detalhados todos efeitos, tal como são efetuados:

- **Chorus**: Adiciona uma onda com frequência ligeiramente superior à atual.

```
1     while i<len(samples):
2         data.append(samples[i] + 32767*sin(2*pi*(freq+30)*i
    /44100))
```

- **Percussão**: Soma uma onda com o quádruplo da frequência, com a sua amplitude a decair ao longo do tempo.

```
1             while i<len(tone['samples']):
2                 data2.append((len(data)-j)*0.1*sin(4*tone['freq'
    ]*2*pi*i/44100))
```

- **Tremolo**: Para variar a amplitude de sinal, este efeito soma a onda atual com uma onda de amplitude mínima multiplicada pelo valor atual da onda.

```
1     while i<len(data):
2         data2.append(data[i] + a*sin(2*pi*freq*i/sample_rate)*
    data[i])
```

- **Dist**: O efeito de distorção deveria elevar os valores da onda atual a um número natural. No entanto, como esse efeito ficaria pouco agradável, multiplicámos os valores da onda atual por ela própria, e depois por uma amplitude mínima.

```
1     while i<len(data):
2     # data[i] = data[i] ** 2
3         data[i] = data[i] * 0.01*data[i]
```

- **Echo**: Este efeito aplica eco a uma música, somando um sinal futuro ao sinal atual, com uma atenuação na amplitude.

```
1     while i<len(data):
2         if i+dura<len(data):
3             data[i+dura] += aten*data[i]
```

- **Envelope**: Este efeito modela nota a nota de acordo com o instrumento. Neste caso, a nota será crescente, de amplitude 0% a 100% no primeiro oitavo tempo da música. No segundo, a nota estabilizará em 50% da sua amplitude. Continuará assim, até ao ultimo oitavo tempo da música, onde a amplitude descrescerá até 0. Caso a frequência da nota anterior seja igual à atual, o efeito anula-se nos dois primeiros oitavos tempos, mantendo-se sempre a 50%. Caso a frequência da nota posterior seja igual à atual, o efeito anula-se nos último oitavos tempos, mantendo-se sempre a 50%.

```
            while i<len(tones[tone]['samples']):
                if i<=len(tones[tone]['samples'])/8:
                    if(tone-1 >= 0 and tones[tone-1]['freq']==
    tones[tone]['freq']):
                        j = 0.5
                    data[k]*=j
                    j += (0.5/(len(tones[tone]['samples'])/8))
    /0.5
                elif i<=2*(len(tones[tone]['samples'])/8):
                    if(tone-1 >= 0 and tones[tone-1]['freq']==
    tones[tone]['freq']):
                        j = 0.5
                    data[k]*=j
                    j -= 0.5/(len(tones[tone]['samples'])/8)
                elif i<=6*(len(tones[tone]['samples'])/8):
                    data[k]*=j
                else:
                    if(tone+1 < len(tones) and tones[tone+1]['
    freq']==tones[tone]['freq']):
                        j = 0.5
                    data[k]*=j
                    j -= 0.5/(2*(len(tones[tone]['samples'])/8))
```

Este módulo teve como principal contribuidor Diogo Ferreira, desenvolvendo este módulo a 100%.

Parte III

Conclusão

Capítulo 4

Conclusão

O resultado final deste projeto é evidenciado na interação utilizador-aplicação, aquilo que aos olhos do utilizador parece ser uma simples página *Web* é o resultado de uma combinação bem estruturada de elementos que interagem entre si e providenciam ao utilizador uma experiência intuitiva. Neste ponto o objetivo do projeto é claramente cumprido. O ponto chave para esta conclusão está assente numa estrutura de código bem planeada o que permite o funcionamento adequado entre os vários elementos usados, nomeadamente as bases de dados, o servidor (*ambiente Python*) e as páginas *Web* (com os respetivos códigos *HTML*, *CSS* e *Scripts*).

É de notar que com poucas ações o utilizador pode chegar às listas de músicas, pode criar uma nova, manipular resultados e ouvir o resultado final. Isto é fruto do cuidado em manter a aplicação intuitiva. É também importante referir o cuidado em manter esteticamente uma apresentação semelhante entre a aplicação *desktop* e a aplicação *mobile*. Também poderiam ser adicionadas posteriormente mais funcionalidades, como apagar uma música, adicionar imagens a músicas ou procurar por letras. Devido ao tempo limitado, não nos foi possível adicionar estas funcionalidade, que facilmen temte podem ser adicionadas posteriormente.

A rapidez na interação utilizador-aplicação *web* é algo que deve ser alvo de comentário. Todos os painéis de navegação *web* são rápidos na sua apresentação, porém o conteúdo variável de página para página já não reflete a rapidez anterior. Isto deve-se ao conteúdo ser invocado por um *script* local que por sua vez invoca uma função do lado do servidor, que interage com a base de dados, que finalmente enviará o conteúdo em bruto para ser processado pelo *script* para ser apresentado ao utilizador. Este processo é moroso principalmente na geração de uma música (em média 5 a 7 segundos).

Foi tido em conta o facto de, durante a escrita de todos os módulos, terem sido adicionados comentários para que o código escrito seja de fácil compreensão Sempre que alguma funcionalidade fosse implementada, era pedido *feedback* a todos os elementos do grupo, para saber se a solução implementada era a mais correta e como podia ser melhorada. Houve grande interação por parte de todos

os membros do grupo, principalmente quando algum tinha dificuldades em escrever alguma parte que lhe era designada. Foram algumas as vezes que o grupo se juntou, onde o desenvolvimento de cada módulo era contribuído por praticamente todos os elementos, algo que não é possível constatar através do sistema de controlo de versões, mas que é oportuno referir. Principalmente nas últimas semanas houve grande cooperação para a finalização da interface *Web* e da aplicação principal. Assim, considera-se cumprido o projeto pedido ainda com alguma margem dentro das datas limite.

Acrónimos

RTTTL Ring Tone Text Transfer Language

JSON JavaScript Object Notation

HTML HyperText Markup Language

HTTP Hypertext Transfer Protocol

CSS Cascading Style Sheets

WAVE WAVEform audio file format

SQL Structured Query Language

UTF-8 8-bit Unicode Transformation Format