PROJETO 2

Universidade de Aveiro

Gonçalo Silva, Samuel Teixeira, Pompeu Costa,
Hugo Hadden

VERSAO FINAL

PROJETO 2

Departamento de Eletrónica, Telecomunicações e Informática

Universidade de Aveiro

Gonçalo Silva, Samuel Teixeira, Pompeu Costa, Hugo Hadden
(103244) goncalolsilva@ua.pt, (103325) samuelsteixeira@ua.pt,
(103294) pompeu@ua.pt,(98449) hugohadden@ua.pt

14/07/2021

Resumo

Este projeto foi realizado no âmbito da cadeira Laboratórios de Informática (LABI) do 1º ano do Mestrado Integrado em Engenharia de Computadores e Telemática (MIECT). Consiste na criação de um sistema que permita criar músicas através da composição de pedaços/ excertos de música. Para além disso, também tivemos de fazer este mesmo relatório em que explicamos o projeto: objetivo, motivação, a metodologia utilizada, resultados, análise e conclusões. Na metodologia, será relatado em pormenor o código feito para construir este projeto , bem como o modo de funcionamento, testagem e comandos git feitos para tal. Nos resultados, será mostrado o fruto de todo o nosso código que é a aplicação web a funcioanr. Por fim, nas conclusões, retira-se o que se alcançou com este projeto, o que aprendemos, o quão útil este projeto é para compreendermos esta matéria da cadeira de LABI e o quão interessante foi realizá-lo.

Agradecimentos

Queremos agradecer a todos os professores da cadeira de LABI por nos terem dado um trabalho interessante, que nos ajudou a compreender os conceitos lecionados nas aulas.

Índice

Capítulo 1

Introdução

Introduz o tema, apresenta a motivação e finalmente a estrutura.

O objetivo deste trabalho é criar uma aplicação web que permita criar músicas através da composição de excertos de música. A Interface web, tem três páginas, na primeira são listadas as músicas existentes, na segunda os excertos e na terceira, um gerador de músicas, que permita ao utilizador criar a sua própria música, baseada nos excertos disponíveis. Além disso, nas duas primeiras páginas, é possível o utilizador visualizar a informação acerca de cada música/ excerto, sendo até possível ouvi-lo.

Este documento está dividido em quatro capítulos. Depois desta introdução, no Capítulo 2 é apresentada a metodologia seguida, no Capítulo 3 são apresentados os resultados obtidos, sendo estes discutidos no Capítulo 4. Finalmente, no Capítulo 5 são apresentadas as conclusões do trabalho.

Capítulo 2

Metodologia

Como o título sugere, neste capítulo vamos mostrar e explicar os métodos e ferramentas que usámos para completar este projeto.

2.1 Gerador de Músicas

Nesta secção será apresentada a metodologia do Gerador de Músicas, ficheiro songEngine.py.

2.1.1 Função readSong

A função readSong, como se poder fer na **Figura 2.1**, recebe como parâmetros um ficheiro .wav e devolve a informação acerca deletion, no formato wave_params, que corresponde a uma lista a cujos índices guardam a informação da seguinte forma:

- **'0'** - Número de canais do ficheiro (nchannels);
- **'1'** - Largura da sample em bytes (sampwidth);
- **'2'** - Frequência da sample em bytes (Framerate);
- **'3'** - Número de frames de audio (nframes);
- **'4'** - Tipo de compressão (comptype)
- **'5'** - Idêntico ao ponto anterior, mas um nome, geralmente 'not compressed' (compname)

2.1.2 Função checkSong

Esta função (**Figura 2.2**) recebe um caminho de um ficheiro e devolve se este existe, em formato de lista. De acordo com o caminho fornecido, as possíveis respostas da função, são:

```python
# @argumentos -> caminho do ficheiro
# @return -> informação acerca do ficheiro
def readSong(filePath):
    checkFile = checkSong(filePath) # Verificar a existência do ficheiro
    if not checkFile[0] :
        return checkFile # Devolver o dicionário de erro

    wf = wave.open(filePath, "rb") # rb = ler em binário

    result = wf.getparams() # quardar os parâmetros
    # wave_read.getparams
    # Returns a namedtuple()
    # (nchannels, sampwidth, framerate, nframes, comptype, compname),
    # equivalent to output of the get*() methods.
    wf.close() # fechar o ficheiro

    return result # devolver os parâmetros
```

Figura 2.1: Código da função readSong

- **'Ficheiro'** - [True, "success"];
- **'Diretório'** - [False, "The provided path is a directory"];
- **'Não encontrado'** - [False, "The provided path doesn't exists"];

```python
# @argumentos -> caminho do ficheiro
# @return -> lista com pos 0 True ou False, se o ficheiro existe ou não
#         -> pos 1, informação de sucesso e erro, caso algum exista
#         -> i.e [True, "sucess"] ou [False, "The provided path doesn't exists"]
def checkSong(filePath):
    if os.path.isfile(filePath) : # Se o caminho fornecido é um ficheiro
        return[True, "success"]
    elif os.path.isdir(filePath) : # Se o caminho fornecido é um diretório
        return[False, "The provided path is a directory"] # retorna uma mensagem de erro
    else : # Se o caminho fornecido não existe
        return [False, "The provided path doesn't exists"] # retorna uma mensagem de erro
```

Figura 2.2: Código da função checkSong

2.1.3 Função durationSong

A função durationSong aceita como parâmetros um caminho de um ficheiro e devolve a sua duração em segundos. Ela efetua este cálculo com a fórmula $waveFile.nframes/float(waveFile.framerate)$ (**Figura 2.3**).

2.1.4 Função calculateFramerate

Na **Figura 2.4** podemos ver a função calculateFramerate que, tendo sido fornecidos os Beats Per Minute (BPM), devolve a framerate pretendida para ajustar na música. utilizando a fórmula $bpm * 44100/60$

```python
# @argumentos -> ficheiro
# @return -> duração música em segundos
def durationSong(filePath):
    checkFile = checkSong(filePath) # Verificar a existência do ficheiro
    if(not checkFile[0]):
            return [checkFile[0], checkFile[1]] # Devolver o dicionário de erro
    w = wave.open(filePath, 'rb') # ler o ficheiro em binário
    return round(w.getnframes() / float(w.getframerate())) #devolver a duração em segundos
```

Figura 2.3: Código da função durationSong

```python
# @argumentos -> beats per minute (bpm)
# @return -> Framerate (Frames per second)
# @formula -> 44100hz = 60 bpm
def calculateFramerate(bpm) :
    # 60 bpm = 1s
    # Usei esta fórmula, pois penso que o valor default de velocidade de cada música é 44110hz
    # 1hz = 1s
    # I.E 61 * 1 / 60 =  +/- 1.1, mas este valor está errado, pois a música teria horas e horas de duração
    # Assim usei uma fórmula que me parece correta
    return bpm * 44100 / 60 # devolve a framerate para utilizar no ficheiro
```

Figura 2.4: Código da função calculateFramerate

2.1.5 Funções fadeInSong e fadeOutSong

A **Figura 2.5** mostra-nos as funções fadeInSong e fadeOutSong que aplicam os efeitos Fade In e Fade Out à música, respetivamente. Para isso, aceita como parâmetros a música, sample rate (ou framerate) e a duração do efeito.

```python
# @argumentos -> música, sample_rate (framerate), duração do efeito
# @return -> música com o efeito de Fade In
def fadeInSong(song, sample_rate, duration) :
    new_song = []
    duration = float(duration)
    time_start = 0
    time_stop = duration * sample_rate
    step = 1.0 / (sample_rate * duration)
    for index, value in enumerate(song):
        time = index
        if time > time_start and time < time_stop :
            new_song.append(value * index * int(step)) # usar o int para não dar erro de conversão com os outos valores
        else :
            new_song.append(value)

    return new_song # devolver nova música com fade-in

# @argumentos -> música, sample_rate (framerate), duração do efeito
# @return -> música com o efeito de Fade out
def fadeOutSong(song, sample_rate, duration) :
    new_song = []
    sample_rate = float(sample_rate)
    duration = float(duration)
    index = 0
    time_start = index - (duration * sample_rate)
    time_stop = index
    step = 1.0 / (sample_rate * duration)

    for index2, value in enumerate(song) :
        time = index2
        if(time > time_start and time < time_stop) :
            new_song.append(value * (index - index2) * step)
        else :
            new_song.append(value)

    return new_song # devolver nova música com fade-out
```

Figura 2.5: Código das funções fadeInSong e fadeOutSong

2.1.6 Função reverseSong

Na **Figura 2.6** está presente a função reverseSong, que aceita uma música e a inverte. Ou seja, o ínicio da música passa a ser o fim e o fim o ínicio.

```
# @argumentos -> música
# @return -> música invertida
def reverseSong(song) :
    new_song = []
    for value in reversed(song): # reverse() inverte a ordem dos valores, com base no índice
        new_song.append(value)
    return new_song #devolver a música invertida
```

Figura 2.6: Código da função reverseSong

2.1.7 Função volumeSong

A **Figura 2.7** mostra a função volumeSong, que ajusta o volume da música, tendo em conta o novo volume fornecido pelo utilizador. Para ajustar o volume da música, a mesma é multiplicada pelo novo volume, em que 1, corresponde ao valor atual, sem alteração, 0.5 diminui o volume e 2 multiplica o volume.

```
# @argumentos -> música e novo volume
# @return -> música com o volume ajustado
def volumeSong(song, new_vol) :
    # Para controlar o volume basta multiplicar todos os valores de amplitude por um factor
    # multiplicativo. Se este factor for 0.5 o volume deverá ser diminuído em metade. Se for
    # 2.0 o volume deverá ser multiplicado por 2
    new_song = []
    factor = float(new_vol)

    for index, value in enumerate(song):
        new_song.append(value * int(factor)) # usei o int, para não existirem erros de conversão com os valores

    return new_song # volume da música alterado
```

Figura 2.7: Código da função reverseSong

2.1.8 Função normalizeSong

Na **Figura 2.8** está presente a função normalizeSong, que aceita uma música e a devolve com o som normalizado/ regulado, fazendo uso da Função volumeSong. No entanto, esta função não funciona na aplicação final, devido a erros de conversão no código.

2.1.9 Função maskSong

Na **Figura 2.9** está presente a função maskSong, que permite aplicar uma máscara à música. Existem três máscaras disponíveis:

- **'silence'** - Permite silenciar a música;
- **'noise'** - Acrescenta ruído à música;

```python
# @argumentos -> música
# @return -> música com o volume normalizado
def normalizeSong(data): # Não está a funcionar corretamente
    new_song = []
    val_max = 32767
    max = 0

    for index, value in enumerate(data):
        if(abs(value)>max):
            max = abs(value)

    new_song = volumeSong(data, val_max / max)

    return new_song # devolve a música normalizada
```

Figura 2.8: Código da função normalizeSong

- **'tone'** - Tonifica a música;

Depois de aplicados os efeitos, a função irá retornar a música com os efeitos aplicados.

```python
# @argumentos -> Musica, sample_rate (framerate), tipo de máscara, começo, duração
# @return -> Música com a máscara aplicada
def maskSong(song, sample_rate, type, start, duration):
    new_song = []
    start = float(start) * sample_rate
    duration = float(duration) * sample_rate
    end = start + duration

    for index, value in enumerate(song):
        # antes estava index > start, mas isso nunca aconteceria e os filtros nunca eram aplicados,
        #  pois o start seria, i.e 2 * 44100 = 88200. e o index, vai de 0...11.. nunca chegaria a esse valor
        if index < start and index < end:
            if(type == 'silence'):
                new_song.append(0)
            elif(type == 'noise'):
                new_song.append(random.randint(-32768, 32767))
            elif(type == 'tone'):
                new_song.append(10000*math.sin(2 * math.pi * 440 * index / sample_rate))
            else:
                new_song.append(value)
        else:
            new_song.append(value)

    return new_song
```

Figura 2.9: Código da função maskSong

2.1.10 Função modulateSong

Na **Figura 2.10** está presente a função modulateSong que aplica uma modulação à música, utilizando uma sample rate e frequência fornecida pelo utilizador. No entanto, esta função não funciona na aplicação final, devido a erros de conversão no código.

```python
# Aplicar uma modulação (multiplicar um som por outro)
# @argumentos -> Musica, sample_rate (Framerate), frequência
# @return -> Música alterada
def modulateSong(song, sample_rate, freq) : # Não está a funcionar corretamente
    new_song = []
    freq = int(freq)
    for index, value in enumerate(song):
        new_song.append(value * math.sin(2 * math.pi * freq * index / int(sample_rate))) # adicionei o int,

    return new_song # devolver a música modulada
```

Figura 2.10: Código da função modulateSong

2.1.11 Função delaySong

A **Figura 2.11** mostra a função delaySong que permite aplicar um delay/ atraso na música, com o início e duração fornecidos pelo utilizador, devolvendo a música com o delay aplicado.

```python
# @argumentos -> Musica, sample_rate (framerate), quantidade, tempo de delay (atraso)
# @return -> Musica com delay
def delaySong(song, sample_rate, amount, delay) :
    amount = float(amount)
    delay = float(delay)

    new_song = [0] * len(song)

    tdelay = delay * sample_rate

    for index, value in enumerate(song):
        if index + int(tdelay) < len(new_song):
            new_song[index] = value
            new_song[index + int(tdelay)] += value * amount
        else:
            new_song[index] = value

    return new_song # devolve Música com um atraso (delay) aplicado
```

Figura 2.11: Código da função delaySong

2.1.12 Função effectsSong

A **Figura 2.12** mostra a função effectsSong, que permite escolher qual o efeito a aplicar a uma música e os parâmetros que serão aplicados.

2.1.13 Função createSong

A **Figura 2.13** mostra a função createSong, onde um dicionário JSON é fornecido como argumento, para criar uma música e depois de os campos deste dicionário serem testados, a função gera uma música com os valores fornecidos. Fazendo uso das funções mencionadas acima, a função também pode aplicar efeitos e alterar parâmetros para combinar vários excertos numa só música, que é devolvida ao utilizador. No entanto, se alguma das verificações ou processos

```python
# Aplicar os efeitos na música
# @argumentos -> Música, bpm (framerate), efeito escolhido
# @return -> Música com o efeito aplicado
def effectsSong(data, bpm, effects):
    if effects == "fadein" : # efeito de Fade In
        data = fadeInSong(data, bpm, 2)
    elif effects == "reverse" : # reverter a música
        data = reverseSong(data)
    elif effects == "normalize" : # normalizar o volume da música
        data = normalizeSong(data) # Não funciona corretamente
    elif effects == "modulate" : # Aplicar uma modulação (multiplicar um som por outro)
        data = modulateSong(data, bpm, 441) # Não funciona corretamente
    elif effects == "delay" : # Introduzir um delay à música
        data = delaySong(data, bpm, 5, 2)
    elif effects == "fadeout" : # efeito de Fade Out
        data = fadeOutSong(data, bpm, 2)
    else :
        print("Effect doesn't exist")

    return data
```

Figura 2.12: Código da função effectsSong

não for bem sucedido, a função devolve uma mensagem de erro em formato de list: [False, "erro"].

2.2 Servidor Cherrypy

2.2.1 Função principal

2.3 SQL

2.4 Interface Web

2.4.1 JavaScript

2.4.2 Gerador de Imagens

2.5 Git

As funcionalidades do git foram muito utilizados neste projeto, desde a simples sincronização de ficheiros e código, até à criação, junção e gestão de branches (**Figura 2.14** e **Figura 2.15**). [1]

2.6 Code UA

As funcionalidades do Code UA forneceram bastante ajuda ao desenvolvimento do projeto, desde a própria visualização dos branches disponíveis, bem como a própria gestão e visualização do código, até à criação de funcionalidades a serem

```python
# @argumentos -> dicionário com informação da música a ser criada
# @return -> lista com pos 0 True ou False, se a música for criada ou não
#         -> pos 1, informação de sucesso e erro, caso algum exista
#         -> 1.e [True, "Song generated"] ou [False, "The provided path doesn't exists"]
def createSong(dictionary):
    # dictionary["samples"][i] access each sample
    for sample in dictionary["samples"] : # certificar que todas as samples existem
        status = checkSong(sample)
        if(not status[0]): # verificar que não houve um erro
            print([status[0], status[1]])
            return [status[0], status[1]] # se houver um erro, devolver

    for music in dictionary["music"] : # verificar a lista das musicas com as samples fornecidas
        for pos in music :
            if pos >= len(dictionary["samples"]) :
                print([False, "Music indexes and samples provided do not match"])
                return [False, "Music indexes and samples provided do not match"]

    data= []
    for i, music in enumerate(dictionary["music"]) :# Percorrer o dicionário das músicas
        if len(music) == 1 : # se não for preciso combinar músicas, apenas se acrescenta ao dicionário
            sample = dictionary["samples"][music[0]]
            w = wave.open(sample, 'rb')
            params = w.getparams() # Obter os parâmetros do excerto
            dataTemp = w.readframes(w.getnframes()) # Ler o excerto
            if i in dictionary["effects"]: # Percorrer a lista de efeitos, para determinar se existe algum
                dataTemp = bytes(effectsSong(dataTemp, calculateFramerate(dictionary["bpm"]), dictionary["effects"][i])) # alterar os dados lidos para o excerto com o efeito aplicado
            data.append([params, dataTemp]) # guardar os dados na lista
            w.close() #fechar o excerto
        elif len(music) > 1 : # se houver mais do que uma música no índice da lista de ficheiros, vamos sobrepô-las
            overlay = b'' # inicializar a variável
            for l, pos in enumerate(music) : # percorrer as músicas presentes no índice
                sample = dictionary["samples"][pos]
                # overlay the samples
                w = wave.open(sample, 'rb')
                params = w.getparams() # Obter os parâmetros do excerto
                dataTemp = w.readframes(w.getnframes()) # Ler o excerto
                if i in dictionary["effects"]: # Percorrer a lista de efeitos, para determinar se existe algum
                    overlay += bytes(effectsSong(dataTemp, calculateFramerate(dictionary["bpm"]), dictionary["effects"][i])) # alterar os dados lidos para o excerto com o efeito aplicado
                else :
                    overlay += dataTemp # copiar o excerto normal

            data.append([params, overlay]) # guardar os dados na lista
        else :
            data.append([0, bytes(0)]) # Adicionar troço de silêncio


    # Ajustar o volume
    data = volumeSong(data, int(dictionary["volume"]) / 5)

    # Adicionar Máscara, caso seje especificado
    if dictionary["mask"] != "none" :
        data[0:][1] = maskSong(data, calculateFramerate(dictionary["bpm"]), dictionary["mask"], 2, 5)

    # sample files are saved in dictionary["samples"]

    # Indices - Valores lista data[0][0]
    # 0 - nchannels
    # 1 - sampwidth
    # 2 - framerate (song pace)
    # 3 - nframes
    # 4 - comptype
    # 5 - compname
    #print(data[0][0]) # Mostrar a informação do ficheiro original DEBUG
    #print("Output Framerate : " + str(calculateFramerate(dictionary["bpm"]))) # Mostrar a frequência do ficheiro de saída DEBUG

    output = wave.open(dictionary["id"], 'wb') # abrir / criar o ficheiro output da música
    output.setparams(data[0][0]) # Escrever os parâmetros do ficheiro original no ficheiro de saída
    output.setframerate(calculateFramerate(dictionary["bpm"])) # Alterar a framerate do ficheiro, tendo em conta os bpm fornecidos
    for l in range(len(data)): # percorrer a música gerada
        output.writeframes(data[i][1]) # escrever no ficheiro de saída
    output.close() # fechar o ficheiro de saída

    print([True, "Song generated"])
    return [True, "Song generated"] # devolver que todas as operações correram com sucesso

createSong(dic) # DEBUG
```

Figura 2.13: Código da função createSong

```
gls@GlS-pc:~/repos/labi2021-p2-g14/goncalo/report-template$ git status
On branch master
Your branch is up to date with 'origin/master'.

nothing to commit, working tree clean
gls@GlS-pc:~/repos/labi2021-p2-g14/goncalo/report-template$ git checkout -b relatorio
Switched to a new branch 'relatorio'
gls@GlS-pc:~/repos/labi2021-p2-g14/goncalo/report-template$ git push origin relatorio
Username for 'https://code.ua.pt': goncalolsilva@ua.pt
Password for 'https://goncalolsilva@ua.pt@code.ua.pt':
Total 0 (delta 0), reused 0 (delta 0)
To https://code.ua.pt/git/labi2021-p2-g14
 * [new branch]      relatorio -> relatorio
```

Figura 2.14: Exemplo da criação de branches (criação branch relatorio)

desenvolvidas e bugs a serem resolvidos. Pode visualizar o projeto no Code UA, através do link: http://code.ua.pt/projects/labi2021-p2-g14 [2]

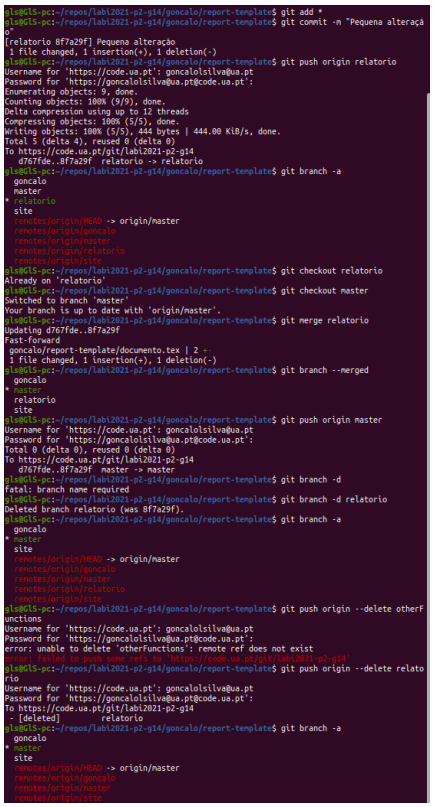

Figura 2.15: Exemplo da eliminação de branches (eliminação branch relatorio)

Capítulo 3

Resultados

Descreve os resultados obtidos com este relatório.

3.1 Funcionamento

A **??** mostra o comando de inicio do servidor. Mostrar o servidor em funcionamento

3.2 Testes

Os testes às funcionalidades foram feitos através de testes funcionais e unitários criados para o efeito. Estes testes são corridos usando a ferramenta pytest incorporada no python. Os testes albergam três ficheiros, o ficheiro test_songEngine.py e test_func_songEngine.py, testam as funções de criação e obtenção de músicas do servidor, através de testes unitários e funcionais. O ficheiro test_app.py, testa as funcionalidadesdo servidor cherrypy.

É possível correr estes testes através do comando **"python3 -m pytest"**, como nos mostra a Figura 3.1.

```
gls@GlS-pc:~/repos/labi2021-p2-g14/goncalo/songs$ python3 -m pytest
============================= test session starts ==============================
platform linux -- Python 3.8.10, pytest-4.6.9, py-1.8.1, pluggy-0.13.0
rootdir: /home/gls/repos/labi2021-p2-g14/goncalo/songs
collected 5 items

test_func_songEngine.py .                                                [ 20%]
test_songEngine.py ....                                                  [100%]

========================== 5 passed in 6.85 seconds ===========================
```

Figura 3.1: Exemplo de execução dos testes ao songEngine

Capítulo 4

Análise

Analisa os resultados. Mostrar que as músicas foram criadas e ficou tudo correto

Capítulo 5

Conclusões

Com este trabalho, conseguimos solidificar o nosso conhecimento em várias linguagens, como HTML, JavaScript, CSS e Python, bem como outras funcionalidades, como o cherrypy, pytest e operações com músicas (wave). Apesar das adversidades, acreditamos que o trabalho foi conseguido com sucesso, criámos uma aplicação web com as funcionalidades referidas e todas as características necessárias para tal, utilizando os recursos que nos foram fornecidos e auxiliando a sua compreensão com este relatório.

Contribuições dos autores

Resumir aqui o que cada autor fez no trabalho. Usar abreviaturas para identificar os autores, por exemplo AS para António Silva. No fim indicar a percentagem de contribuição de cada autor.

Acrónimos

MIECT Mestrado Integrado em Engenharia de Computadores e Telemática

LABI Laboratórios de Informática

BPM Beats Per Minute

Bibliografia

[1] *Git*, `https://git-scm.com/`.

[2] *CODE UA*, `https://code.ua.pt/`.