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 do mesmo, 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

Neste secção será apresentada a metodologia do Cherrypy, ficheiro App.py.

O servidor contêm 2 funções de ajuda, funções para abrir as diversas páginas web e funções que as páginas web chamam para receber/introduzir dados da base da dados.

2.2.1 Função getSample

A função getSample é uma função de ajuda, que tem um parâmetro (id) o qual é usado para procurar um excerto. Esta devolve o caminho do sistema de ficheiros,caso o excerto exista. Para tal faz uma pesquisa na base de dados, se a pesquisa devolver alguma coisa então o excerto existe, caso contrário o excerto não existe então devolve None.
A **Figura 2.14** mostra como isto é feito em código.

2.2.2 Função getSong

A função getSong é a outra função de ajuda. Esta, tal como a anterior, tem um parâmetro (id) que é usado para procurar uma música. O processo é exatamente igual ao da função anterior mas a pesquisa é feita na tabela Musicas em vez de ser na tabela Samples.
A **Figura 2.15** mostra como isto é feito em código.

```
# @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

```
def getSample(id):
    db = sql.connect(DB_NAME)

    try:
        result = db.execute(
            "SELECT id FROM Samples WHERE id = ?", (str(id),)).fetchone()[0]

        return "samples/" + str(result) + ".wav"

    except:
        return None
    finally:
        db.close()
```

Figura 2.14: Código da função getSample

```
def index(self):
    cherrypy.response.headers["Content-Type"] = "text/html"
    return open("index.html", "r", encoding="utf-8")
    try:
        result = db.execute(
            "SELECT id FROM Musicas WHERE id=?", (str(id),)).fetchone()[0]

        return "songs/" + str(result) + ".wav"

    except:
        return None
    finally:
        db .close()
```

Figura 2.15: Código da função getSong

2.2.3 Funções para abrir páginas web

O site tem várias páginas e para navegar nelas é preciso que o servidor esteja pronto para receber os pedidos de navegação. Para tal existem funções que devolvem a página desejada. Como estas funções são todas iguais ou bastante parecidas, decidimos juntá-las numa secção.
A **Figura 2.16** mostra como isto é feito em código.

```
def index(self):
    cherrypy.response.headers["Content-Type"] = "text/html"
    return open("index.html", "r", encoding="utf-8")
```

Figura 2.16: Código da função index

2.2.4 Função list

A função list devolve uma lista com músicas ou samples dependendo do que é pedido. Para tal, esta faz uma pesquisa na base de dados e devolve a lista com os resultados. Caso o tipo pedido não seja válido, a função devolve um dicionário a indicar que deu erro e com uma mensagem a explicar.
A **Figura 2.17** mostra como isto é feito em código.

2.2.5 Função get

A função get devolve o caminho do sistema de ficheiros da música ou excerto. Para tal, usa as funções de ajuda **Subseção 2.2.1** e **Subseção 2.2.2**. Se estas funções devolverem alguma coisa, então a música/excerto existe e o caminho deste é devolvido, caso contrário devolve um dicionário com a indicação de que

```python
def list(self, type):
    cherrypy.response.headers["Content-Type"] = "text/json"
    if str(type).lower() == "songs":
        db = sql.connect(DB_NAME)
        result = db.execute(
            "SELECT * FROM Musicas").fetchall()

        db.close()
        return json.dumps(result)
    elif str(type).lower() == "samples":
        db = sql.connect(DB_NAME)
        result = db.execute("SELECT * FROM Samples").fetchall()
        db.close()
        return json.dumps(result)
    else:
        return json.dumps({"result": "failure", "erro": "type invalido"})
```

Figura 2.17: Código da função list

falhou e uma mensagem de erro.
A **Figura 2.18** mostra como isto é feito em código.

```python
def get(self, id):
    result = getSong(id)
    cherrypy.response.headers["Content-Type"] = "text/json"
    if result != None:
        return json.dumps({"result": "sucess", "path": result})

    # nao foi encontrada nenhuma musica com o id especificado
    # possivelmente um excerto?
    result = getSample(id)

    if result != None:
        return json.dumps({"result": "sucess", "path": result})

    return json.dumps({"result": "failure", "erro": "nao existe excerto nem musica com o id"})
```

Figura 2.18: Código da função get

2.2.6 Função put

A função put gera uma música e insere a informação na base de dados. A música gerada é baseada numa pauta criada em js numa das páginas web.
Devolve um dicionário a indicar o resultado da operação (sucesso ou falho) e uma mensagem de erro caso tenha falhado.
A **Figura 2.19** mostra como isto é feito em código.

```python
def put(self, pauta, nome, autor):
    h = sha256()
    h.update((str(nome) + str(autor)).encode("utf-8"))
    n_id = h.hexdigest()
    jPauta = json.loads(pauta)
    song = getSong(id)

    cherrypy.response.headers["Content-Type"] = "text/json"

    if song != None:
        return json.dumps({"result": "failure", "erro": "autor ja tem uma musica com esse nome"})

    jPauta["id"] = "songs/" + n_id + ".wav"
    created = []
    try:
        created = createSong(jPauta)
    except:
        return json.dumps({"result": "failure", "erro": "erro interno"})
    if not created[0]:
        return json.dumps({"result": "failure", "erro": "erro"})

    length = durationSong("songs/" + n_id + ".wav")
    # adicionar a db
    sqlCommand = "INSERT INTO Musicas (id,nome,autor,length,date,votos) VALUES (?,?,?,?,?,?)"
    db = sql.connect(DB_NAME)
    db.execute(sqlCommand, (n_id, nome, autor,
                            length, str(date.today()), 0))
    db.commit()
    db.close()

    return json.dumps({"result": "sucesso"})
```

Figura 2.19: Código da função put

2.2.7 Função uploadSample

A função uploadSample deixa o utilizador introduzir um excerto no sistema, através de uma página web. Na função, o excerto é lido e um ficheiro wav é criado com um nome (id) também gerado em código. Os dados do ficheiro são guardados na base de dados caso o excerto tenha sido guardado com sucesso.
Devolve um dicionário a indicar o resultado da operação e uma mensagem de erro caso tenha falhado.
A **Figura 2.20** mostra como isto é feito em código.

2.2.8 Função vote

A função vote atualiza os votos, de uma música, na base de dados.
Devolve um dicionário a indicar o resultado e uma mensagem de erro caso a operação tenha falhado.
A **Figura 2.21** mostra como isto é feito em código.

2.3 SQL

Para a criação da base de dados, utilizamos o terminal, usando o programa sqlite3. Usando os comandos(CREATE TABLE Musicas();), colocando os di-

```python
def uploadSample(self, sample, nome):
    cherrypy.response.headers["Content-Type"] = "text/json"
    h = sha256()
    h.update(str(nome).encode("utf-8"))
    id = h.hexdigest()

    s = getSample(id)

    if s != None:
        return json.dumps({"result": "failure", "erro": "ja existe um excerto com esse nome"})
    try:
        uploadPath = os.path.join(PATH, "samples")
        uploadFile = os.path.join(uploadPath, id + ".wav")

        open(uploadFile, "wb").write(sample)
        length = durationSong("samples/" + id + ".wav")
    except:
        return json.dumps({"result": "failure", "erro": "ocorreu um erro durante o processamento"})

    db = sql.connect(DB_NAME)
    try:
        db.execute(
            "INSERT INTO Samples(id,nome,length) VALUES(?,?,?)", (id, str(nome), length))
        db.commit()
    except:
        return json.dumps({"result": "failure", "erro": "ocorreu um erro a inserir na db"})
    finally:
        db.close()

    return json.dumps({"result": "sucess"})
```

Figura 2.20: Código da função uploadSample

ferentes campos com a respetiva indicaçao do tipo(INTEGER, TEXT, FLOAT, etc) dentro dos parênteses. Neste caso usámos vários campos, Id para identicar a música, Nome para indicarmos o nome da música, Autor para sabermos a quem pertence a música, path para podermos armazenar e saber onde está a música, votos para podermos saber quantos votos a música já recebeu, length para sabermos o tamanho da música e por fim date para sabermos quando é que a música foi criada. Durante a resolução do trabalho foram colocados alguns valores para as funcionalidades do trabalho serem testadas, para isso usámos o comando INSERT disponível no sqlite3 (INSERT INTO Musicas VALUES();) introduzindo os valores para cada campo dentro dos parênteses (**Figura 2.22**). Também criámos uma tabela Samples para guardar informação sobre os excertos.

sectionInterface Web Nesta secção, será explicada a estrutura e funcionamento da Interface Web e o modo como comunica com o servidor.

2.3.1 Página Web

A Interface foi criada através de um template retirado da internet (Spify). Tal como é mencionado no enunciado, é constituída por 3 páginas, sendo a primeira a página onde são listadas todas as músicas da aplicação, a segunda a página onde são listados os excertos e a terceira a página que tem o gerador de músicas.

```python
def vote(self, id, points):
    cherrypy.response.headers["Content-Type"] = "text/json"
    db = sql.connect(DB_NAME)
    num_votes = None
    try:
        num_votes = db.execute(
            "SELECT votos FROM Musicas WHERE id = ?", (str(id),)).fetchone()[0]
    except:
        db.close()
        return json.dumps({"result": "failure", "erro": "musica nao encontrada"})

    try:
        points = int(points)
    except:
        return json.dumps({"result": "failure", "erro": "points nao e inteiro"})

    if points != -1 and points != 1:
        return json.dumps({"result": "failure", "erro": "points tem de ser 1 ou -1"})

    # atualiza o numero de votos e atualiza a tabela
    new_votes = int(num_votes) + points
    db.execute("UPDATE Musicas SET votos = ? WHERE id = ?",
               (new_votes, str(id),))

    db.commit()  # funciona como o git commit e git push
    db.close()
    return json.dumps({"result": "success"})
```

Figura 2.21: Código da função vote

```
gls@GlS-pc:~/repos/labi2021-p2-g14/pompeu/projeto$ sqlite3 BaseDados.db
SQLite version 3.31.1 2020-01-27 19:55:54
Enter ".help" for usage hints.
sqlite> .tables
Musicas  Samples
sqlite> .schema
CREATE TABLE Musicas(id text,nome text,autor text,length integer,date text,votos
 integer);
CREATE TABLE Samples(id text,nome text,length integer);
sqlite>
```

Figura 2.22: Tabelas da base de dados

Index

A página Index sofreu algumas alterações, nomeadamente a cor de fundo, a imagem de overlay, a barra de navegação, alguns detalhes como títulos, o rodapé e a adição da funcionalidade. Foi criado um design para a aparência das músicas, com os respetivos dados: autor, áudio, data, dar votos e mostrar o número de votos. Esta aparência foi criada sem o uso de BS (bs), à exceção das divisões de localização deste design.

Samples

A página Samples era a antiga página "Articles"do template original. Esta página sofreu as mesmas alterações que a página Index, mudando apenas a funcionalidade da página. Para criar esta última alteração, foi criada uma tabela com a ajuda da biblioteca bs; onde é possível ver o autor do excerto e o áudio deste mesmo.

Gerador

A página Gerador era a antiga página "Privacy", sofreu as mesmas alterações que as restantes, à exceção da funcionalidade dessa página. Foi criada uma interface para o Gerador de músicas à base de bs, onde se pode selecionar alguns parãmetros localizados nessa interface de maneira a ser gerada uma música com essas características.

2.3.2 JavaScript

O template já tinha algum código JS (js) dentro de uma pasta denominada "js", o qual não foi alterado. Porém, foi adicionado a essa pasta 3 ficheiros: "index_funcs, getdata e musicgen". Esses ficheiros são responsáveis pelas funcionalidades das 3 páginas, respetivamente.

O ficheiro index_funcs tem como objetivo adicionar à página Index as funcionalidades correspondentes à apresentação das músicas do sistema. Foram criadas 1 array "songs", em que cada posição correspondea um array com informações sobre as músicas, e 1 dicionário "dic"que serve de auxílio para a função que guardar os estados de voto quando se volta a entrar na página.

Função setMusic

A função setMusic tem como objetivo criar a interface que apresenta todas as opções descritas para a página Index. São criadas os respetivos items(div, h, span, etc...) através de "document.createElement("item")", adicionando as respetivas classes através de "item.classList.add("class")", e algumas possíveis marcas style ou outros atributos através de "item.atributo". Foi também estabeleciada a relação filial correta dos determinados items através de "item.appendChild(itemfilho)". De seguida, a última linha de código serve para estabelecer o estado dos botões de votação, sendo ambos falsos inicialmente porque ainda não houve votação.

Funções likeUp e dislike

Estas funções definem em que circunstâncias é que houve uma votação positiva ou negativa. Ambas possuem um algoritmo muito idêntico, mudando apenas os ícones que são colocados. Na função likeUp, a variável like corresponde à marca "a"que é responsável pelo ícone de votação positiva. Se em dic, na posição

0(correspondente ao valor do voto positivo) estiver false, houve voto positivo e é criada a marca correspondente, sendo guardado em sessionStorage esse dado no formato JSON (json). Caso tenha havido voto negativo, é executada a função correspondente(dislike). Se na posição 0 estiver true, não houve voto positivo e é alterado o ícone. Se não houve alteração de votos, é guardado em sessionStorage.pref o valor none que indica que não houve alterações. A função de dislike é praticamente igual, como referido, alterando apenas os ícones e as condições, que passam a corresponder ao voto negativo.

Função loadPref

Para que os votos do utilizador sejam guardados na aplicação, sem que sejam perdidos caso o utilizador saia desta, é necessária que haja algum mecanismo para ser feito o que foi descrito, neste caso é loadPref. Se não houve alterações de votação, não é feito nada. Caso tenha havido alterações de votação, são analizadas essas alterações e, no caso da existência de voto positivo, é executada a função likeUp e é executada a função dislike no caso do voto negativo.

Função getMusic

Esta função é a principal e obtem os dados pertencentes ao servidor e executa também as restantes funções. A constante result e a constante data guardam esses dados no formato json para poderem ser manipulados. A variável songs vai guardar os valores em data. Logo a seguir, são executadas as funções setMusic e loadPref.

O ficheiro getdata tem como objetivo adicionar à página Excertos as funcionalidades correspondentes à apresentação dos excertos do sistema.

userAction

Esta função tem como objetivo realizar todas as funcionalidades da responsabilidade de getdata.js. Para isso, é criado, à semelhança de getMusic, uma constante response possui os dados do servido, que são convertidos para json e guardados em myJson. A constante sampArr vai guardar o valor da chave samples proveniente de myJson, que contem os excertos. A contante tbl corresponde à tabela que se encontra na página. De seguida, serão criadas as várias interfaces de cada música existente. A constante name indica o nome da música e encontra-se na chave nome do dicionário da determinada posição do array sampArr. Será criada uma marca table row (tr) e as suas 2 células, que são a que indicará o nome do autor do excerto e o excerto em si. Por fim, é estabelecida a relação filial entre estas marcas e a tabela (tbl).

Por fim, resta-nos explicar o ficheiro musicGen. Este ficheiro tem como objetivo gerar a interface gráfica do gerador de músicas correspondente à página Gerador do site.

Função generateLines

De acordo com o meu colega, Esta função tem como objetivo criar a tabela dinâmica da interface do gerador de músicas, Será criada uma tabela com as respetivas marcas, as suas classes (bs) e é estabelecida a filiação respetiva.

Função getSampleList

Esta função fornece os excertos necessários ao gerador de músicas. A constante response vai guardar os excertos provenientes do servidor e a constante myJson vai guardar os dados de response no formato json, com o objetido de serem fornecidas as samples para posteriormente executar a função generateLines.

Função gerarArray

A função gerarArray transforma as checkboxes num array em que cada posição contem um array com as opções de cada checkboxe. Fim

Função getEffects

Esta função fornece um array com os efeitos aplicados em cada uma das linhas. Se uma linha não possuir efeitos, o array não vai introduzir nada na posição correspondente a essa linha.

Função getSamplesWithPath

Esta função cria um array com os caminhos dos excertos para serem utilizados no servidor para gerar a música.

Função gerar

Esta função envia os dados necessários para o servidor gerar uma música e avisa logo de seguida se essa ação foi sucedida ou não. São criadas constantes correspondentes aos variados dados (volume, BPM, efeitos, etc...), para além do array com as opções das checkboxes(é chamada a função gerarArray), dos efeitos e dos excertos (são executadas as funções gerarArray, geteffects e getSamplesWithPath, respetivamente, para estes 3 últimos). Estes dados são guardados numa constante, convertidos para json e enviados neste formato para o servidor.

2.4 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 2.23** e **Figura 2.24**). [1]

```
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 2.23: Exemplo da criação de branches (criação branch relatorio)

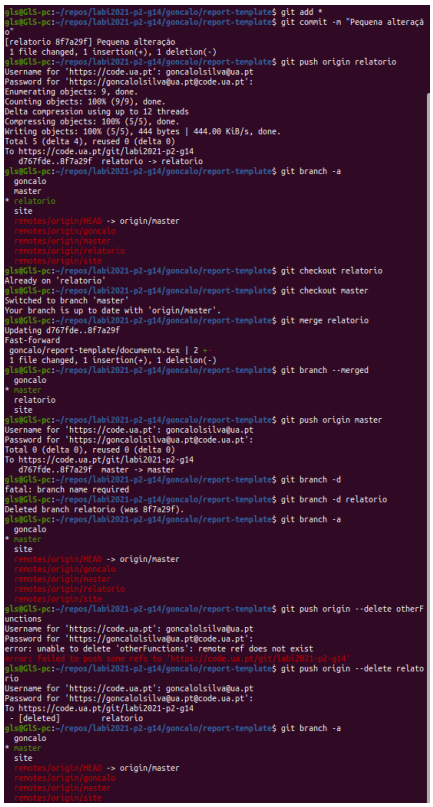

Figura 2.24: Exemplo da eliminação de branches (eliminação branch relatorio)

2.5 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 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]

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.

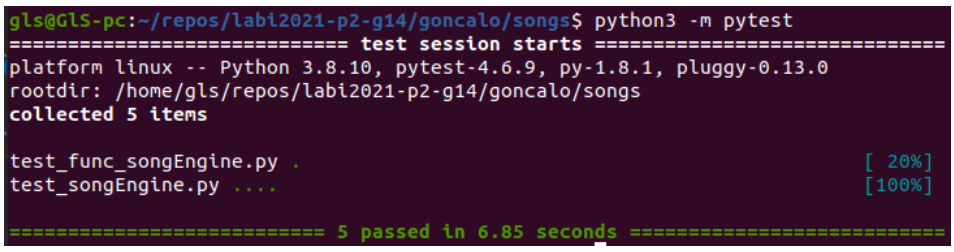

Figura 3.1: Exemplo de execução dos testes ao songEngine

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

js JS
JavaScript

bs BS
Twitter Bootstrap

json JSON
JavaScript Objective Notation

Bibliografia

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

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