PROJETO FINAL

Universidade de Aveiro

Afonso Baixo, Luís Leal, Paulo Macedo

PROJETO FINAL

Departamento de Eletrónica, Telecomunicações e Informática

Universidade de Aveiro

Afonso Baixo, Luís Leal, Paulo Macedo
(108237) afonso.baixo@ua.pt, (103511) lecl@ua.pt,
(102620) paulomacedo@ua.pt

CodeUA: `https://code.ua.pt/projects/labi2022g6`

11 de Julho de 2022

Resumo

Este projeto consite no desenvolvimento de um sistema que irá funcionar como uma plataforma (**Editora**) para colecionar imagens, ou seja, algo similar a uma caderneta de cromos.

O utilizador poderá assumir um de dois papéis disponíveis, o de **Curador** e o de **Colecionador**, sendo que o primeiro poderá apenas fazer *upload* de imagens para o sistema e o segundo requisitar e trocar imagens com outros utilizadores, podendo apenas requisitar imagens que estejam livres.

A interface com a qual o utilizador vai interagir é constituída por páginas Hypertext Markup Language (HTML) e permite ao utilizador visualizar quais as imagens disponíveis na plataforma e a que coleção pertencem, sendo que existem imagens livres e outras não requisitáveis pois já pertencem a outro utilizador. A alternativa é requisitar imagens livres ou propôr uma troca de imagens. É também possível ao utilizador visualizar todos os "cromos"da sua coleção.

Em relação ao processamento de imagem, este será feito sempre que um **Curador** fizer *upload* de uma imagem para a plataforma pois esta necessita de ser redimensionada de forma a ser exposta num formato definido como padrão para que depois possa ser visualizada corretamente na plataforma. Importa também referir que sempre que uma imagem é requisitada por um **Colecionador**, esta passará a ter uma "marca de água"composta pelo username do próprio, o que a torna não disponível para outros utilizadores.

Conteúdo

1 Introdução **1**

2 Componentes **2**
2.1 Interface Web 2
2.2 Aplicação Web 2
2.2.1 Função user_is_logged 2
2.2.2 Função db_store_image 3
2.2.3 Funções get 3
2.2.4 Função draft_image 4
2.2.5 Função db_update_image_owner 4
2.2.6 Classe CreateImage 5
2.2.7 Classe Users 5
2.2.8 Classe Cromos 6
2.2.9 Classe Root 8
2.3 Persistência 9
2.4 Processador de Imagens 10

3 Resultados **11**

4 Conclusões **13**

Lista de Figuras

2.1 Função user_is_logged 3
2.2 Função db_store_image 3
2.3 Funções get 4
2.4 Função draft_image 4
2.5 Função db_update_image_owner 5
2.6 Classe CreateImage 5
2.7 Classe Users 6
2.8 Classe Cromos 8
2.9 Classe Root 9
2.10 database.py 9
2.11 Função watermark 10

3.1 É possível criar uma coleção atribuindo um nome à mesma e adicionando uma imagem (a); É possível ver todas as imagens disponíveis na coleção (b) 11
3.2 São apresentadas todas as imagens da coleção (a) com a opção de ver as informações (b) 12
3.3 Resultados - 3 12

Capítulo 1

Introdução

O objetivo deste trabalho é colocar em prática os conhecimentos adquiridos aos longo do semestre na Unidade Curricular (UC) Laboratórios de Informática (LabI), tais como a implementação de aplicações e serviços Web, bases de dados e manipulação de imagem, através da criação de uma plataforma digital de colecionismo de imagens digitais, ou seja, uma caderneta de cromos.

Este projeto ficará a funcionar no servidor XCOA, através do link disponibilizado. O sistema proposto tem de ser composto pelos seguintes componentes:

- Interface Web
- Aplicação Web
- Persistência
- Processador de Imagens

A Interface Web (Seção 2.1) é constituída por 8 páginas HTML, fornecendo a interface de interação utilizador/sistema.

A Aplicação Web (Seção 2.2) consiste num programa desenvolvido em Python que serve conteúdos estáticos, apresentando métodos que permitem navegar entre os componentes e fornece uma interface programática que permite obter e inserir informação relativa às imagens e aos autores.

A componente da Persistência (Seção 2.3) é composta por métodos que permitem o registo e obtenção de informação através de uma Base de Dados (BD) relacional desenvolvida em Python com recurso à biblioteca SQLite.

Por fim, o Processador de Imagens (Seção 2.4), será um componente integrado na Aplicação Web que faz a gestão das imagens, mais em concreto, o seu armazenamento e obtenção.

Capítulo 2

Componentes

Neste capítulo serão descritos e analisados os vários componentes, referidos anteriormente, que constituem este projeto.

2.1 Interface Web

A Interface Web é composta por JavaScript (JS), HTML, Cascading Style Sheets (CSS) e Bootstrap, sendo os dois últimos mais utilizados para realizar alterações a nível estético.

O JS é um dos grandes pilares deste projeto, pois a sua utilização juntamente com a ferramenta *Ajax* permite a implementação dos métodos *GET* e *POST*, que têm como objetivo a comunicação com o *back-end* através de dicionários JavaScript Object Notation (JSON). *POST* é o método utilizado para enviar informação enquanto que o método *GET* permite receber informação. Para além da ferramenta *Ajax*, é também utilizado o JS para descrever o funcionamento de botões implementados no código HTML, com recurso ao Bootstrap.

2.2 Aplicação Web

Nesta secção será abordada a aplicação web, desenvolvida em Python, sendo descrito o funcionamento das principais classes e funções utilizadas no programa *web.py*.

2.2.1 Função user_is_logged

Esta função é invocada aquando do *login* por parte do utilizador. O seu papel passa por armazenar o *username* e o *token* em *cookies*, permitindo um acesso instantâneo às informações do utilizador.

```python
def user_is_logged(request):
    with sqlite3.connect(DB) as conn:
        cursor = conn.cursor()

        try:
            username = request.cookie["username"]
            token = request.cookie["token"]
        except KeyError:
            return False

        cursor.execute("SELECT token from users WHERE username=?", [
                       username.value])

        row = cursor.fetchone()
        return row and row[0] != None and row[0] == token.value
```

Figura 2.1: Função user_is_logged

2.2.2 Função db_store_image

Função utilizada para armazenar as imagens e os seus respetivos dados à BD e criar uma nova coleção caso esta não exista.

```python
def db_store_image(collection_name, file, username):
    with sqlite3.connect(DB) as conn:
        cursor = conn.cursor()

        # retornar id do utilizador
        cursor.execute("SELECT id FROM users WHERE username=?;", [username])
        username_id = cursor.fetchone()[0]

        # verificar se a collection ja existe
        cursor.execute("SELECT id_collection FROM collections WHERE name=?;", [
                       collection_name])
        row = cursor.fetchone()

        collection_id = -1
        if row:
            collection_id = row[0]
        else:
            # adicionar colecao
            cursor.execute("INSERT INTO collections(name) VALUES(?);", [
                           collection_name])
            collection_id = cursor.lastrowid

        cursor.execute("INSERT INTO images(img_name, collection_id, creation_date, uploaded_by) VALUES(?, ?, ?, ?);",
                       [file.filename, collection_id, datetime.now(timezone.utc), username_id])

        return (collection_id, cursor.lastrowid)
```

Figura 2.2: Função db_store_image

2.2.3 Funções get

Todas as funções *get* presentes no programa *web.py* são utilizadas para obter informações específicas relativas ao utilizador, às imagens e às coleções armazenadas na BD *cromos.db*.

```python
def get_image_name(id):
    with sqlite3.connect(DB) as conn:
        cursor = conn.cursor()
        cursor.execute("SELECT img_name FROM images WHERE image_id=?", [id])

        return cursor.fetchone()[0]


def get_collections():
    with sqlite3.connect(DB) as conn:
        cursor = conn.cursor()
        cursor.execute("SELECT * FROM collections;")
        return cursor.fetchall()


def get_collection_image(id):
    with sqlite3.connect(DB) as conn:
        cursor = conn.cursor()
        cursor.execute(
            "SELECT img_name FROM images WHERE collection_id=?;", [id])
        return cursor.fetchone()[0]


def get_images_from_collection(id):
    with sqlite3.connect(DB) as conn:
        cursor = conn.cursor()
        cursor.execute("SELECT * FROM images WHERE collection_id=?;", [id])
        return cursor.fetchall()


def get_user(id):
    with sqlite3.connect(DB) as conn:
        cursor = conn.cursor()
        cursor.execute("SELECT username FROM users WHERE id=?", [id])

        return cursor.fetchone()[0]


def get_user_id(username):
    with sqlite3.connect(DB) as conn:
        cursor = conn.cursor()
        cursor.execute("SELECT id FROM users WHERE username=?", [username])
        row = cursor.fetchone()

        if not row:
            raise cherrypy.HTTPError(404)

        return row[0]
```

(a)

```python
def get_collection_name(id):
    with sqlite3.connect(DB) as conn:
        cursor = conn.cursor()
        cursor.execute(
            "SELECT name FROM collections WHERE id_collection=?;", [id])
        return cursor.fetchone()[0]


def get_image_transactions(id):
    with sqlite3.connect(DB) as conn:
        cursor = conn.cursor()
        cursor.execute("SELECT * FROM transactions WHERE image_id=?;", [id])
        return cursor.fetchall()


def get_image_information(id):
    username = cherrypy.request.cookie["username"].value

    ret = {"id": id}
    with sqlite3.connect(DB) as conn:
        cursor = conn.cursor()
        cursor.execute("SELECT * FROM images WHERE image_id=?;", [id])
        row = cursor.fetchone()

        if row[5]:
            ret["owner"] = get_user(row[5])
        else:
            ret["owner"] = ""

        ret["able_to_transfer"] = get_user_id(username) == row[5]
        ret["creation_date"] = row[3]
        ret["uploaded_by"] = get_user(row[4])
        ret["collection_name"] = get_collection_name(row[2])
        ret["img_name"] = row[1]
        ret["img_url"] = f"/static/images/{row[1]}"

        image_transactions = get_image_transactions(id)
        ret["transactions"] = []

        for transaction in image_transactions:
            ret["transactions"].append({
                "ts": transaction[3],
                "owner": get_user(transaction[1])
            })

        return ret
```

(b)

Figura 2.3: Funções get

2.2.4 Função draft_image

Sempre que um utilizador requisitar uma imagem, esta função será invocada com o propósito de atualizar as informações da imagem, na BD, de acordo com os dados do novo proprietário. Numa tabela chamada **transactions**, são inseridas informações os sobre a transação da imagem, ou seja, o *id* do utilizador, a data de transação e o *id* da imagem em questão.

```python
def draft_image(id):
    username = cherrypy.request.cookie["username"].value

    with sqlite3.connect(DB) as conn:
        cursor = conn.cursor()

        user_id = get_user_id(username)
        cursor.execute(
            "UPDATE images SET owner_id=? WHERE image_id=?", [user_id, id])

        cursor.execute("INSERT INTO transactions(current_owner_id, ts, image_id) VALUES(?, ?, ?);",
                       [user_id, datetime.now(timezone.utc), id])
```

Figura 2.4: Função draft_image

2.2.5 Função db_update_image_owner

Numa situação em que o utilizador pretenda fazer uma troca de imagens com outro utilizador, esta função será invocada e irá atualizar, as informações, relativas à imagem, presentes na BD, na tabela *images*, de forma a atribuir um novo *owner*. Quanto à tabela *transactions*, esta será atualizada com as

informações mais relevantes relativas à troca, tais como a data, o seu *owner* atual e anterior e o *id*.

```
def db_update_image_owner(id, new_owner):
    username = cherrypy.request.cookie["username"].value

    with sqlite3.connect(DB) as conn:
        cursor = conn.cursor()

        # atualizar owner da imagem
        owner_id = get_user_id(new_owner)
        cursor.execute(
            "UPDATE images SET owner_id=? WHERE image_id=?;", [owner_id, id])

        # adicionar mais uma transferencia
        cursor.execute("INSERT INTO transactions(current_owner_id, previous_owner_id, ts, image_id) VALUES(?, ?, ?, ?);",
                       [owner_id, get_user_id(username), datetime.now(timezone.utc), id])
```

Figura 2.5: Função db_update_image_owner

2.2.6 Classe CreateImage

A classe **CreateImage** é utilizada para o funcionamento do *upload* de uma imagem, obtendo o nome da imagem e o seu ficheiro que serão utilizados na função **db_store_image** de modo a armazenar esta nova imagem.

```
class CreateImage(object):

    @cherrypy.expose
    @cherrypy.tools.json_out()
    def create(self, name=None, file=None):
        destination = os.path.join("public/images/", file.filename)
        with open(destination, 'wb') as f:
            shutil.copyfileobj(file.file, f)

        username = cherrypy.request.cookie["username"].value

        collection_id, image_id = db_store_image(
            name, file, username)

        return {
            "id": collection_id,
            "image_id": image_id
        }
```

Figura 2.6: Classe CreateImage

2.2.7 Classe Users

Dentro da classe **Users**, podemos encontrar funções relacionadas com a autenticação, criação e gestão de utilizadores.

Na função **auth**, após o *login* por parte do utilizador, os seus dados vão ser comparados com os dados presentes na BD (**cromos.db**) e, caso correspondam, é-lhe atribuído um *token*. Este *token* é gerado aleatóriamente e é composto por 8 caracteres American Standard Code for Information Interchange (ASCII).

A função **create** é utilizada para registar os dados do utilizador na BD e a função **valid** retorna as *strings* "valid"ou "invalid"dependendo se o utilizador está ou não conectado.

Por fim, a função **profile** é utilizada de forma a atualizar o perfil do utilizador com as imagens que este requisitou.

```
class Users(object):
    @cherrypy.expose
    @cherrypy.tools.json_out()
    @allow('POST')
    def auth(self, username=None, password=None):
        with sqlite3.connect(DB) as conn:
            cursor = conn.cursor()
            cursor.execute(
                "SELECT username, password FROM users WHERE username=?", [username])
            query_result = cursor.fetchone()

            if not query_result or username != query_result[0] or password != query_result[1]:
                cherrypy.response.status = 404
                return {
                    "authentication": "failed"
                }

            letters = string.ascii_letters
            token = ''.join(random.choice(letters) for i in range(8))
            cursor.execute("UPDATE users SET token=? WHERE username=?", [
                           token, username])

            cherrypy.response.status = 200
            return {
                "authentication": "OK",
                "token": token
            }

    @cherrypy.expose
    @cherrypy.tools.json_out()
    @allow('POST')
    def create(self, username=None, password=None):
        if not username or not password:
            cherrypy.response.status = 400
            return

        with sqlite3.connect(DB) as conn:
            cursor = conn.cursor()

            cursor.execute(
                "SELECT username FROM users WHERE username=?", [username])
            query_result = cursor.fetchone()

            if query_result:
                cherrypy.response.status = 409
                return

            cherrypy.response.status = 200
            cursor.execute("INSERT INTO users(username, password) VALUES (?,?)", [
                           username, password])
        return "success"
```

(a)

```
@cherrypy.expose
@cherrypy.tools.json_out()
def valid(self):
    if user_is_logged(cherrypy.request):
        return "valid"
    return "invalid"

@cherrypy.expose
@cherrypy.tools.json_out()
def profile(self):
    username = cherrypy.request.cookie["username"].value

    ret = []
    with sqlite3.connect(DB) as conn:
        cursor = conn.cursor()
        cursor.execute("SELECT img_name FROM images WHERE owner_id=?", [
                       get_user_id(username)])
        images = cursor.fetchall()

        for image_name in images:
            ret.append({
                "username": username,
                "img_name": image_name[0],
                "img_path": f"static/images/{image_name[0]}"
            })

    return ret
```

(b)

Figura 2.7: Classe Users

2.2.8 Classe Cromos

A classe **Cromos** é constituída pelas funções **index**, **draft**, **image** e **transfer**. Na primeira, caso não seja atribuído nenhum *id*, a sua funcionalidade é armazenar todas as informações das imagens e das respetivas coleções num dicionário JSON que será enviado para o *front-end*; no caso de ser atribuído um *id*,

os efeitos serão os mesmos, mas para uma imagem específica. Na segunda, são invocadas as funções **draft_image** e **watermark**, mencionadas anteriormente, que servirão para auxiliar o processo de requisição. A terceira serve o propósito de obter as informações completas de uma imagem associada a um *id*. Por último, a função **transfer**, invoca as funções **db_update_image_owner** e **watermark**, para serem utilizadas no processo de troca de imagem entre utilizadores.

```python
class Cromos(object):
    def __init__(self):
        self.name = CreateImage()

    @cherrypy.expose
    @cherrypy.tools.json_out()
    def index(self, id=None):
        if not user_is_logged(cherrypy.request):
            return "unauthorized"

        if id:
            image_rows = get_images_from_collection(int(id))

            ret = []
            for row in image_rows:
                image_owner = None
                if row[5]:
                    image_owner = get_user(row[5])

                image = {
                    "id_image": row[0],
                    "img_path": f"static/images/{row[1]}",
                    "name": row[1],
                    "owner": image_owner
                }

                ret.append(image)
            return ret

        ret = []
        collection_rows = get_collections()

        for row in collection_rows:
            collection_image = get_collection_image(row[0])
            collection = {
                "id_collection": row[0],
                "name": row[1],
                "img_path": f"static/images/{collection_image}"
            }

            ret.append(collection)

        return ret
```

(a)

```python
@cherrypy.expose
@cherrypy.tools.json_out()
def draft(self, id=None):

    if not user_is_logged(cherrypy.request):
        raise cherrypy.HTTPError(401)

    if not id:
        raise cherrypy.HTTPError(500)

    destination = os.path.join("./public/images/", get_image_name(id))
    username = cherrypy.request.cookie["username"].value

    draft_image(id)
    watermark(destination, username)

    return "Draft made successfully"

@cherrypy.expose
@cherrypy.tools.json_out()
def image(self, id=None):
    if not user_is_logged(cherrypy.request):
        raise cherrypy.HTTPError(401)

    if not id:
        raise cherrypy.HTTPError(500)

    return get_image_information(int(id))

@cherrypy.expose
@cherrypy.tools.json_out()
def transfer(self, id=None, new_owner=None):
    destination = os.path.join("./public/images/", get_image_name(id))

    db_update_image_owner(int(id), new_owner)
    watermark(destination, new_owner)

    return "Transfer made successfully"
```

(b)

Figura 2.8: Classe Cromos

2.2.9 Classe Root

A função **__init__** classe **Root** permite a comunicação direta entre o *back-end* e o *front-end* utilizando dicionários JSON.

As restantes funções utilizadas na **Root** servem para retornar o conteúdo HTML.

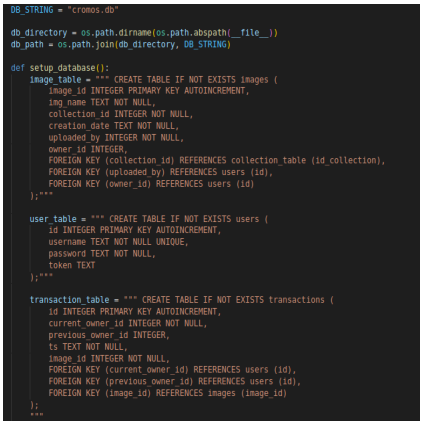
(a)

(b)

Figura 2.9: Classe Root

2.3 Persistência

A BD é composta por 4 tabelas relacionais, sendo estas denominadas por *users*, *images*, *collections* e *transactions*. Nas tabelas *images*, *transactions* e *collections*, são feitas referências às chaves da tabela *users* e, desta forma, é possível fazer a ligação entre as tabelas através do *id* dos utilizadores e do *id* da coleção.



(a)

(b)

Figura 2.10: database.py

2.4 Processador de Imagens

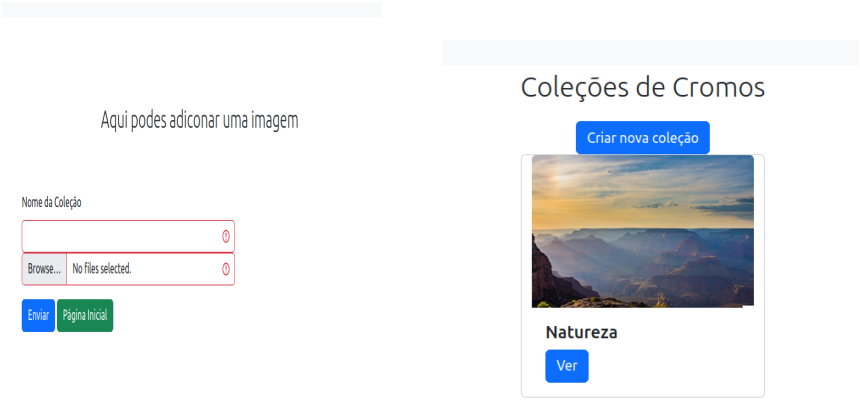
Quanto ao processamento de imagem, este vai ser feito através da função **watermark**, presente no programa *web.py*. Esta função é responsável por colocar uma marca de água, composta pelo *username* do utilizador que fez a requisição, na imagem requisitada, de forma a poder identificar o seu atual proprietário. Esta imagem vai ser guardada no directório *images*.

```
def watermark(destination, owner):
    # Create an Image Object from an Image
    im = Image.open(destination)
    width, height = im.size

    typeFont = "./public/fonts/arial.ttf"
    draw = ImageDraw.Draw(im)
    font = ImageFont.truetype(typeFont, 15)

    w, h = font.getsize(owner)
    pos = (width - w + 10, height - h + 5)


    rectangle_pos = (width - w - 100, height - h - 10, width, height)
    draw.rectangle(rectangle_pos, fill="white")

    draw.text(pos, owner, fill="black", anchor="mm", font=font) # Marca

    im.save(destination)
```

Figura 2.11: Função watermark

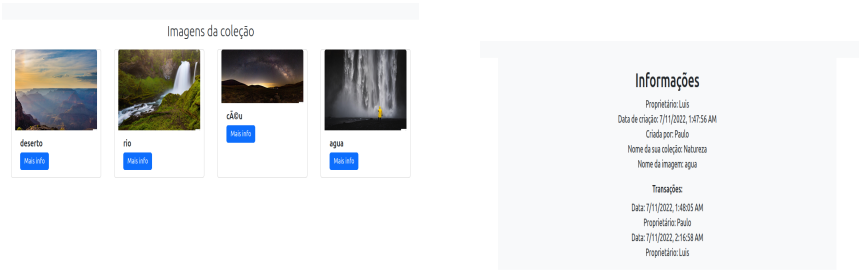
Capítulo 3

Resultados



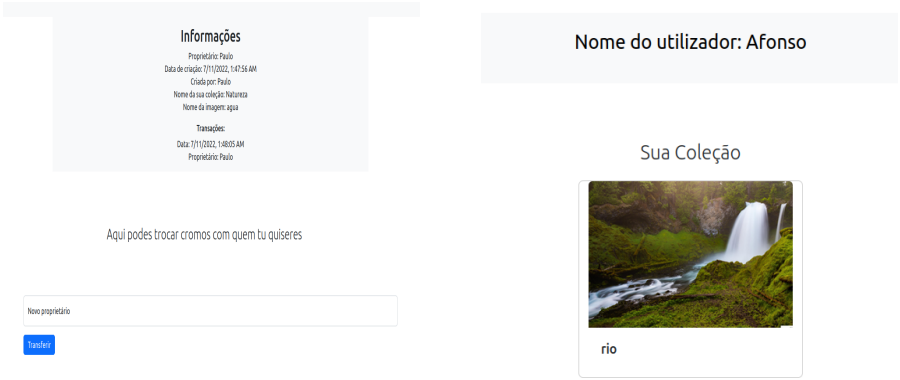
(a) (b)

Figura 3.1: É possível criar uma coleção atribuindo um nome à mesma e adicionando uma imagem (a); É possível ver todas as imagens disponíveis na coleção (b)



(a) (b)

Figura 3.2: São apresentadas todas as imagens da coleção (a) com a opção de ver as informações (b)



(a) (b)

Figura 3.3: Também é possível transferir a imagem, caso seja o dono da mesma (a); Perfil do utilizador onde é apresentado o nome e a sua coleção pessoal (b)

Capítulo 4

Conclusões

Graças a este projeto, foi possível adquirir e consolidar conhecimentos fundamentais para o desenvolvimento de aplicações web.

É importante referir a utilização de JS, HTML, CSS e Bootstrap para o desenvolvimento do *front-end* e Python, JSON, jQuery, Ajax e SQLite, utilizados para desenvolver o *back-end*. Estas "ferramentas"foram fundamentais para o projeto e são excelentes recursos para utilizar em trabalhos futuros.

Contribuições dos autores

O Afonso Baixo (AB) contribuiu para o *web.py* , JS e ficou responsável pelo *debugging*. O Luís Leal (LL) contribuiu para o *web.py*, *database.py*, CSS e para o relatório. O Paulo Macedo (PM) contribui para o *web.py*, *database.py*, HTML, JS e CSS.

Afonso Baixo, Luís Leal, Paulo Macedo: 30%, 20%, 50%

Acrónimos

UC Unidade Curricular

LabI Laboratórios de Informática

AB Afonso Baixo

LL Luís Leal

PM Paulo Macedo

JS JavaScript

JSON JavaScript Object Notation

HTML Hypertext Markup Language

CSS Cascading Style Sheets

ASCII American Standard Code for Information Interchange

BD Base de Dados

Bibliografia

[1] Bootstrap.

[2] Bootstrap icons.

[3] Cherrypy documentation.

[4] Flaticon.

[5] jquery.

[6] Sqlite documentation.

[7] Sqlite3 documentation.

[8] Stack overflow.

[9] W3schools.