Tema 5

Aplicações e Serviços Web

Objetivos:

- Servidores Web
- Serviços Web

5.1 Introdução

A World Wide Web (WWW) ou *Web* como é hoje popularmente conhecida, teve a sua génese em 1990 no Conseil Européen pour la Recherche Nucléaire (CERN) pelas mãos de Tim Berners-Lee. Inicialmente, a *Web* pretendia ser um sistema de hiper-texto que permitisse aos cientistas seguir rapidamente as referências num documento, evitando o processo tedioso de apontar e pesquisar referências. A *Web* pretendia na altura ser um repositório de informação estruturado em torno de um grafo (daí a *Web*), em que o utilizador pudesse seguir qualquer percurso entre os diversos documentos interligados pelas suas referências.

A *Web* assenta em 3 tecnologias, já tratadas nesta disciplina:

· Um sistema global de identificadores únicos/uniformes (URL, URI).

· Uma linguagem de representação de informação (HTML).

· Um protocolo de comunicação cliente/servidor (HTTP).

Com base nestas tecnologias, podemos não só transferir ficheiros estáticos entre um servidor e um cliente equipado com um *Web browser*, como também podemos construir documentos de forma dinâmica a partir de dados recebidos ou disponíveis no servidor num determinado momento.

Um bom exemplo deste último caso são os serviços meterológicos que processam dados de observação e produzem documentos JavaScript Object Notation (JSON)[1] e Extensible Markup Language (XML) com as observações e previsões, para consumo por humanos ou outras máquinas. Este guião irá guiar o desenvolvimento de uma aplicação *Web* dinâmica com base na linguagem de programação *Python* e no formato de documentos JSON.

5.2 Servidores *Web*

5.2.1 Common Gateway Interface

Um servidor *Web* é uma aplicação de software que permite a comunicação entre dois equipamentos através do protocolo HTTP. A função inicial de um servidor *Web* era a de fornecer documentos armazenados em disco em formato HyperText Markup Language (HTML)[2] a um cliente remoto equipado com um *Web* browser. Esta simples tarefa desde cedo demonstrou-se demasiado restritiva, uma vez que frequentemente era necessário condicionar os dados nos documentos HTML a vários fatores, tais como: a identidade do utilizador, a sua localização, a sua língua nativa, etc.

É desta forma que surge o conceito de Common Gateway Interface (CGI). A CGI permite ao servidor interagir com um programa externo capaz de produzir dinamicamente conteúdos de qualquer formato. O *standard* CGI define um conjunto de parâmetros que são passados do servidor *Web* para a aplicação externa (denominada script CGI), assim como o formato que essa mesma aplicação deve de obedecer por forma ao servidor *Web* re-interpretar o seu *output* antes de enviar ao *Web* browser do cliente.

É possível criar programas CGI em qualquer linguagem, inclusive uma linguagem de *scripting* como a *Bash*.

Exercício 5.1

No servidor **`xcoa.av.it.pt`**, no diretório **`public_html`**, crie um novo diretório com o nome **`cgi-bin`**. Dentro desse diretório crie um novo ficheiro **`test.sh`** com o seguinte conteúdo:

```
#!/bin/bash
echo "Content-type: text/plain"
echo ""
echo "Hello World"
```

Dê permissões de execução ao ficheiro (**`chmod +x test.sh`**).
Execute-o na linha de comando (**`./test.sh`**).

Agora no seu navegador *Web*, aceda ao ficheiro que acabou de criar.
Altere o ficheiro para mostrar outras *Strings*.

Do exercício anterior é importante reter a necessidade do programa imprimir um cabeçalho com informação do tipo de ficheiro que será criado dinamicamente. Através da interface CGI é possível não só criar ficheiros de texto (*plain*, HTML, JS, etc) como também ficheiros binários (imagens, vídeos, etc).

Exercício 5.2

Altere o ficheiro anterior adicionando o comando **env**.

```
#!/bin/bash
echo Content-type: text/plain
echo ""
echo "Hello World"
env
```

Aceda ao ficheiro no seu navegador *Web*.

O resultado deste exercício mostra as *variáveis de ambiente* que o servidor *Web* envia para o programa externo através da interface CGI.[1]

5.2.2 Servidores Aplicacionais

Um servidor *Web* é uma aplicação de software que permite a comunicação entre dois equipamentos através do protocolo HTTP. A função inicial de um servidor *Web* era a de fornecer documentos armazenados em disco em formato HTML a um cliente remoto equipado com um *Web* browser.

Esta simples tarefa desde cedo demonstrou-se demasiado restritiva, uma vez que frequentemente era necessário condicionar os dados nos documentos HTML a vários fatores, tais como: a identidade do utilizador, a sua localização, a sua língua nativa, etc.

Servidores aplicacionais como o *Glassfish*, *JBoss*, *.NET* permitem ao programador ultrapassar muitas destas dificuldades ao incorporarem em si próprios código desenvolvido por programadores externos. Não estamos mais na situação de o servidor fornecer um ficheiro estático, mas na de o próprio programa incluir o servidor *Web* e poder fornecer conteúdos gerados de forma programática.

Neste capítulo, vamos abordar um servidor aplicacional específico para *Python*. O *CherryPy* é um servidor aplicacional simples mas poderoso, usado tanto para pequenas aplicações como para grandes serviços (ex.: *Hulu*, *Netflix*).

[1] As variáveis de ambiente (*environment variables*) são um mecanismo providenciado pelo sistema operativo para disponibilizar informação aos programas, para além do mecanismo de passagem de argumentos.

O *CherryPy* pode ser usado sozinho (*stand-alone*) ou através de um servidor *Web* tradicional via interfaces Web Server Gateway Interface (WSGI). Nesta disciplina, vamos usar o *CherryPy* apenas como servidor *stand-alone*. Para instalar o *CherryPy* pode recorrer ao gestor de pacotes da sua distribuição Linux ou ao **`pip`**.

No ubuntu pode executar:

```
sudo apt-get install python3-cherrypy3
```

Em alternativa pode executar:

```
sudo pip3 install CherryPy  ou  sudo pip3 install --upgrade CherryPy
```

É altamente aconselhado que se instale o *CherryPy* via o comando `pip` pois a versão é mais recente.

Para instalar o pacote **`cherrypy`** no *Windows* deve invocar uma *Power Shell* em modo de administrador e executar o comando **`pip install cherrypy`**.

O *CherryPy* é composto por 8 módulos:

`CherryPy.engine` Controla o início e o fim dos processos e o processamento de eventos.

`CherryPy.server` Configura e controla a WSGI ou o servidor HTTP.

`CherryPy.tools` Conjunto de ferramentas para processamento de um pedido HTTP.

`CherryPy.dispatch` Conjunto de *dispatchers* que permitem controlar o encaminhamento de pedidos para os *handlers*.

`CherryPy.config` Determina o comportamento da aplicação.

`CherryPy.tree` A árvore de objetos percorrida pela maioria dos *dispatchers*.

`CherryPy.request` O objeto que representa o pedido HTTP.

`CherryPy.response` O objeto que representa a resposta HTTP.

Comecemos por criar uma aplicação semelhante ao *script* CGI seguinte. Revendo cada linha deste exemplo, começamos por identificar a importação do módulo *CherryPy*. De seguida temos a declaração de uma classe *HelloWorld*. Esta classe é composta por um método chamado *index* que devolve uma *String*. O *decorador* **`@cherrypy.expose`** determina que o método *index* deverá ser exposto ao cliente *Web*. Por fim, o módulo *CherryPy* cria um objeto da classe *HelloWorld* e inicia um servidor com ele.

Exercício 5.3

Crie no seu próprio computador o seguinte ficheiro *Python*.

```
import cherrypy

class HelloWorld(object):
    @cherrypy.expose
    def index(self):
        return "Hello World!"

cherrypy.quickstart(HelloWorld())
```

Sendo que neste caso a aplicação é lançada automaticamente quando existirem alterações ao ficheiro e permite que seja terminada usando **`CTRL-C`**.

No seu *Web browser* aceda à aplicação usando o endereço `http://localhost:8080/`.

Quando um cliente *Web* acede ao servidor aplicacional *CherryPy*, este procura por um objeto e método que possa atender ao pedido do cliente. Neste exemplo básico, existe apenas um objeto e método que irá servir ao cliente a *String* `"Hello World"`. O *CherryPy* disponibiliza através do **`CherryPy.request.headers`** as variáveis enviadas pelo cliente ao servidor.

Exercício 5.4

Altere o programa anterior para mostrar o nome do servidor ao qual o cliente fez um pedido HTTP.

```
...
        host = cherrypy.request.headers["Host"]
        return "You have successfully reached " + host
```

Qualquer objeto associado ao objeto raiz é acessível através do sistema interno de mapeamento *URL*-para-objeto. Ou seja, definindo funções que implementem uma lógica, é possível expor essas funções, de forma quase automática, através de um Uniform Resource Locator (URL)[3], acessível por um *browser*.

No entanto, tal não significa que um objeto esteja exposto na *Web*. É necessário que o objeto seja exposto explicitamente como visto anteriormente.

Mais uma vez, é importante reter alguns aspetos do exercício anterior. O método **`index`** serve os conteúdos na raiz do URL (/) e cada método tem que ser exposto individualmente.

Exercício 5.5

Crie um novo programa com o seguinte conteúdo

```
import cherrypy

class Node(object):
    @cherrypy.expose
    def index(self):
        return "Eu sou o índice do Node (Node.index)"

    @cherrypy.expose
    def page(self):
        return "Eu sou um método do Node (Node.page)"

class Root(object):
    def __init__(self):
        self.node = Node()

    @cherrypy.expose
    def index(self):
        return "Eu sou o índice do Root (Root.index)"

    @cherrypy.expose
    def page(self):
        return "Eu sou um método do Root (Root.page)"

if __name__ == "__main__":
    cherrypy.quickstart(Root(), "/")
```

Aceda a cada um dos recursos a partir do seu navegador *Web* (**/**, **/page**, **/node** e **/node/page**). Pode alterar as mensagens devolvidas de forma a verificar que o conteúdo é dinâmico.

Em alternativa, pode usar o módulo **psutil** para devolver estatísticas do sistema.

Exercício 5.6

Acrescente agora uma nova classe **HTMLDocument** que devolva o conteúdo de um ficheiro HTML designado, por exemplo, por **documento.html**.

Ou seja, o programa irá abrir um ficheiro existente (**open**) ler o seu conteúdo e devolver os dados lidos (**return**). Não se esqueça de associar um novo objecto designado, por exemplo, por **html** na raiz da sua aplicação para invocar esta classe.

Aceda ao novo recurso a partir do seu navegador *Web* (**/html**).

5.2.3 Formulário HTML

O protocolo HTTP define dois métodos principais para a troca de informação entre cliente e servidor: os métodos **GET** e **POST**.

O método **GET** já foi extensivamente usado nos capítulos e secções anteriores, e permite ao cliente *Web* solicitar um documento que resida no servidor *Web*.

Por sua vez o método **POST** permite enviar informação do cliente *Web* para o servidor *Web*. É geralmente usado para enviar ao servidor um ficheiro ou um formulário HTML preenchido.

Exercício 5.7

Crie uma página HTML para o formulário (ficheiro designado por **formulario.html**) com o código seguinte:

```
<form action="actions/doLogin" method="post">
    <p>Username</p>
    <input type="text" name="username" value="" size="15" maxlength="40"/>
    <p>Password</p>
    <input type="password" name="password" value="" size="10" maxlength="40"/>
    <p><input type="submit" value="Login"/></p>
    <p><input type="reset" value="Clear"/></p>
</form>
```

Crie este novo método **form** na raiz da sua aplicação:

```
    @cherrypy.expose
    def form(self):
        cherrypy.response.headers["Content-Type"] = "text/html"
        return open("formulario.html")
```

Aceda ao novo recurso a partir do seu navegador *Web* (**/form**) e verifique que quando tenta submeter o formulário (ao clicar no botão `Login`) o navegador dá um erro, mais concretamente o erro `(404, "The path '/actions/doLogin' was not found.")`.

Isto porque a submissão do formulário de *login* necessita ainda da implementação do método `doLogin` que deve ser associado ao objeto `actions`.

Importa referir que os argumentos *username* e *password* chegam até à aplicação *Web* através de um mapeamento direto do nome das variáveis do formulário HTML para os argumentos do método **doLogin** (também mapeado diretamente).

Exercício 5.8

Acrescente a nova classe **Actions** que implementa o método **doLogin**. Não se esqueça de associar o novo objecto **actions** na raiz da sua aplicação para invocar esta classe.

```
class Actions(object):
    @cherrypy.expose
    def doLogin(self, username=None, password=None):
        return "Verificar as credenciais do utilizador " + username
```

Abra o formulário através do endereço `http://localhost:8080/form/`, preencha-o, submeta-o e verifique que agora o navegador já não dá erro.

Altere a funcionalidade do método para verificar se o utilizador e a senha corresponde a um utilizador específico acrescentando à mensagem indicada a informação de "Acesso concedido" ou "Acesso negado".

5.3 Introdução aos Serviços Web

Na secção anterior viu-se como um cliente *Web* pode interagir com uma aplicação *Web* alojada no servidor. Nesta secção irá abordar-se como duas aplicações podem interagir entre si através do protocolo HTTP.

O primeiro desafio que se coloca é como escrever uma aplicação *Python* capaz de aceder a uma página *Web* via o protocolo HTTP. Para tal vamos fazer uso da biblioteca **requests**, cuja documentação completa encontra-se disponível em `http://docs.python-requests.org/en/master/`.

A biblioteca **requests** permite-nos aceder a uma página *Web* de forma muito semelhante à que utilizamos em *Python* para aceder a um ficheiro.

```
import requests

f = requests.get("http://www.python.org")
print(f.status_code)
```

O uso directo do método **get** permite-nos obter o conteúdo de um recurso HTTP através do método **GET**. No entanto, se pretendermos enviar algum conteúdo para uma aplicação *Web*, é necessário usar o método **POST** como vimos anteriormente.

Exercício 5.9

Faça um pedido **GET** ao endereço `http://www.ua.pt`. A sua aplicação deverá ler por completo o conteúdo da página da Universidade de Aveiro.

Imprima para a consola dados relevantes como os cabeçalhos da resposta ou, por exemplo, o tipo de conteúdo (**`headers['Content-Type']`**), que deverá ser texto `html` codificado em "utf-8".

Utilizando o módulo `time` pode determinar qual é o tempo necessário para obter a página. Pode igualmente testar com outros ficheiros maiores, como por exemplo os disponíveis na página do *kernel Linux*.

O método **`POST`** possibilita o envio de informação codificada no corpo do pedido **`POST`**. A codificação dos dados segue um de dois *standards* definidos pelo World Wide Web Consortium (W3C), o **`application/x-www-form-urlencoded`** e o **`multipart/form-data`**.

O primeiro formato é o usado por omissão e permite o envio de informação trivial como variáveis não muito extensas. O segundo é apropriado para o envio de variáveis mais extensas assim como de ficheiros.

O módulo utilizado realiza esta formatação por defeito, enviando um dicionário qualquer que seja fornecido.

```
import requests

url = ...
values = {"nome": "Ana", "idade": 20}
r = requests.post(url, data=values)
print(r.status_code)
```

Exercício 5.10

Fazendo uso da aplicação *Web* desenvolvida anteriormente, que continha um formulário, implemente uma aplicação capaz de fazer *login*.

Os exercícios anteriores demonstraram como criar uma aplicação *Web* capaz de interagir com um cliente (*Web browser*), mas a sua utilidade pode ser transposta para a comunicação entre duas aplicações.

Exercício 5.11

O *OpenStreetMaps* dispõe de uma Application Programming Interface (API) que permite converter um endereço em coordenadas (latitude e longitude). Neste exercício deverá usar a API do OpenStreetMaps com base no seguinte código para encontrar as coordenadas de qualquer cidade passada ao seu programa através de um argumento de linha de comando.

```
# Morada da Universidade de Aveiro
address = "Universidade de Aveiro, 3810-193 Aveiro, Portugal"

servurl = "https://nominatim.openstreetmap.org/search.php?format=json&q=%s" % address

r = requests.get(servurl)
```

Verifique o método `json()` do resultado do pedido. Como o pedido indica que o formato deverá ser JSON, a resposta está disponível nesse método.

Imprima as coordenadas e a restante informação obtida

5.4 Introdução aos Conteúdos Estáticos

Nos exercícios anteriores permitimos ao nosso servidor aplicacional servir uma página HTML com o conteúdo de um ficheiro usando um método manual. Repare que o método não é escalável, pois é necessário abrir, ler e devolver todos os ficheiros necessários.

Uma alternativa é definir regras para servir ficheiros individuais, o que é apresentado no exemplo que se segue:

```
import os
import cherrypy

PATH = os.path.abspath(os.path.dirname(__file__))

conf = {
        "/documento": {
                        "tools.staticfile.on": True,
                        "tools.staticfile.filename": os.path.join(PATH, "documento.html")
                      }
    }
...
cherrypy.quickstart(Root(), "/", config=conf)
```

Exercício 5.12

Crie um programa que devolva um ficheiro, mas que o faça através de uma configuração do próprio servidor.

Podemos também automatizar este processo indicando ao **Cherrypy** que todos os conteúdos presentes num determinado diretório são estáticos. O exemplo que se segue considera que existe um diretório chamado `static`, localizado no mesmo diretório do programa *Python*, sendo que todo o seu conteúdo é estático, sendo servido automaticamente.

```
import os
import cherrypy

PATH = os.path.abspath(os.path.dirname(__file__))

conf = {
        "/static": {
            "tools.staticdir.on": True,
            "tools.staticdir.dir": os.path.join(PATH, "static")
        },
}
...

cherrypy.quickstart(Root(), "/", config=conf)
```

Exercício 5.13

Implemente o exemplo anterior de forma a servir o mesmo ficheiro que usou anteriormente, mas de forma automática. Crie entradas adicionais na configuração de forma a ter ficheiros Cascading Style Sheets (CSS)[4], JavaScript (JS)[5] e ou imagens, também eles estáticos, cada um no seu diretório específico.

5.5 Servidor WEB com Conteúdos Estáticos e Dinâmicos

5.5.1 Introdução

Uma aplicação Web dinâmica é normalmente composta pelos elementos que se apresentam na Figura 5.1. Assim temos um programa desenvolvido na linguagem de programação *Python* neste caso designado por **app.py** para permitir recorrer a conteúdos dinâmicos e/ou a bases de dados para ter persistência. Tem também os diretórios seguintes: **html** para as páginas (HTML) que definem a estrutura da aplicação, entre elas a página de arranque que habitualmente se designa por **index.html**; **css** para os ficheiros de estilos que configuram o aspeto das páginas; **js** para os ficheiros de **JavaScript** que dão dinâmica às páginas; e **images** para as figuras e ícones que são carregadas nas páginas.

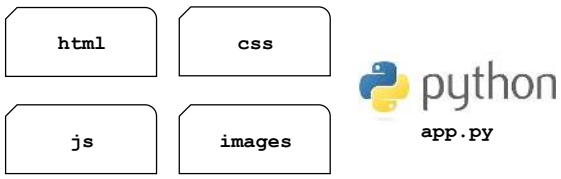

Figura 5.1: Projeto Web.

Poderá ter eventualmente outros diretórios. Por exemplo, um diretório para armazenar ficheiros de audio e/ou de imagem carregados pela aplicação.

De maneira a ter acesso aos diversos recursos da aplicação é conveniente definir os caminhos para estes diretórios estáticos e assim servir automaticamente os ficheiros constituintes da aplicação. O *CherryPy* permite definir uma variável de configuração (que vamos designar por **config**), que a partir do diretório atual onde a aplicação é executada (designado por **baseDir**), cria um dicionário com os caminhos para todos os diretórios constituintes da aplicação, tal como se apresenta de seguida (programa **app.py**).

```
import os.path
import cherrypy

# The absolute path to this file's base directory
baseDir = os.path.dirname(os.path.abspath(__file__))

# Dictionary with this application's static directories configuration
config = {
    "/":        { "tools.staticdir.root": baseDir },

    "/html":    { "tools.staticdir.on": True,
                  "tools.staticdir.dir": "html" },

    "/js":      { "tools.staticdir.on": True,
                  "tools.staticdir.dir": "js" },
```

```
    "/css":     { "tools.staticdir.on": True,
                  "tools.staticdir.dir": "css" },

    "/images":  { "tools.staticdir.on": True,
                  "tools.staticdir.dir": "images" },
}

class Root:
    @cherrypy.expose
    def index(self):
        return open("html/index.html")

cherrypy.quickstart(Root(), "/", config)
```

A aplicação começa por expor o método **index** que se encontra no diretório **html** e que serve como página de entrada da aplicação Web na raiz do URL (/) e a partir da qual todas as restantes páginas poderão ser acedidas pelo utilizador da aplicação.

Exercício 5.14

No sítio da disciplina existe um arquivo ZIP chamado **projeto.zip**. Obtenha este ficheiro, coloque-o dentro do diretório da disciplina e extraia o seu conteúdo.

Entre no diretório **projeto** e compare o seu conteúdo (diretórios e ficheiros) com a Figura 5.1. Veja os ficheiros existentes em cada diretório.

De seguida execute a aplicação Web (**python3 app.py**).

No navegador *Web* aceda à aplicação através do endereço **http://localhost:8080/**.

Navege na aplicação e experimente todos os conteúdos dinâmicos da página principal (gráficos, mapa e manipulação da imagem).

Experimente navegar diretamente na página principal e verifique que ela não carrega todos os recursos ficando com um aspeto diferente devido à falta dos estilos e da ligação à funcionalidade disponibilizada pelo código **JavaScript**.

Aceda também à página acerca (**about.html**). Aceda ainda ao TópicoD - UpLoad (página **upload.html**) e verifique que esta página ainda está incompleta uma vez que não faz nada.

Agora que já experimentou toda a funcionalidade presente na aplicação vamos acrescentar alguns conteúdos dinâmicos. Os dois primeiros serão acrescentados no início da página **index.html** no Tópico A. E o terceiro será acrescentado à página **upload.html**.

5.5.2 Acrescentar conteúdos dinâmicos

Vamos começar por um exemplo simples como é o caso de obter a data e hora. Para isto serão necessários os seguintes componentes:

- Um botão para refrescar a informação (**Date & Time Refresh**).
- Código *JavaScript* que obtenha a informação do servidor.
- Uma aplicação que implemente o serviço pedido.
- Um elemento HTML para armazenar o resultado.

Os dois elementos HTML são adicionados no elemento de classe **content_clock** (marca ** **), colocando o seguinte excerto de código no ínicio do Tópico A, logo a seguir ao comentário **<!- Relógio ->**.

```
<!-- Relógio -->
<div class="content_clock">
    <button id="refresh_clock" class="btn btn-block btn-primary">Date & Time Refresh</button>
        <div style="border: 2px solid black;">
        <div id="clock" style="width:100%; margin: 0px; padding: 0px; text-align:center;"></div>
    </div>
</div>
```

Crie um ficheiro designado por **clock.js** com o seguinte código **JavaScript**. Neste caso, o código espera que seja enviado um elemento JSON com dois atributos: **date** e **time**. O pedido é activado quando o elemento com identificador **refresh_clock** receber um evento de **click**. Não se esqueça de incluir no cabeçalho da página **index.html** a referência ao ficheiro **clock.js**.

```
function clock() {
    $.get("/time",
          function(response) {
            var text="<h2>"+response.date+"</h2><br /><h2>"+response.time+"</h2>";
            $("#clock").html(text);
          });
}

$( document ).ready(function() {
    $("#refresh_clock").on("click", clock);
});
```

Do lado do serviço é necessário implementar um método que devolva a data e a hora. Acrescente à classe **Root** da aplicação **app.py** o método **time** com o seguinte código. Tem também que importar o pacote **time** no início da aplicação.

```
...
import time
...

class Root:
    ...

    # Relógio
    @cherrypy.expose
    def time(self):
        cherrypy.response.headers["Content-Type"] = "application/json"
        return time.strftime('{"date":"%d-%m-%Y", "time":"%H:%M:%S"}').encode("utf-8")
```

Exercício 5.15

Faça as alterações sugeridas. Relance a aplicação e teste esta nova funcionalidade acionando o botão de *refresh*.

O segundo exemplo pretende devolver algo mais útil, tal como a distância para o estádio do SL Benfica (38.752667, -9.184711). Ou de outro qualquer estádio à sua escolha bastando para esse efeito saber as respetivas coordenadas (latitude e longitude).

Precisamos também de dois elementos HTML. Eles são adicionados no elemento de classe **`content_dst`**, colocando o seguinte excerto de código no Tópico A, logo a seguir ao comentário **`<!- Distância ->`**.

```
<!-- Distância -->
<div class="content_dst">
    <button id="refresh_dst" class="btn btn-block btn-primary">Distance to Stadium Refresh</button>
        <div style="border: 2px solid black;">
            <div id="dst" style="width:100\%; margin: 0px; text-align:center;"></div>
        </div>
</div>
```

Também precisa de criar um ficheiro designado por **`dst.js`** com o seguinte código **`JavaScript`**. Neste caso, o código espera que seja enviado um elemento JSON com apenas um atributo que é a distância representada por **`dst`**.

O pedido é activado quando o elemento com identificador **`refresh_dst`** receber um evento de **`click`**. Nessa altura a função **`refresh`** vai fazer um pedido de geolocalização para saber as coordenadas do utilizador. Não se esqueça de incluir no cabeçalho da página **`index.html`** a referência ao ficheiro **`dst.js`**.

```
function dst(position){
    $.get("/dst",
          { lat: position.coords.latitude, lon: position.coords.longitude },
          function(response){
            var text="<h2> Estádio dos Lampiões a "+response.distance+" km</h2>";
            $("#dst").html(text);
          });
}

function refresh(){
    navigator.geolocation.getCurrentPosition(dst);
}

$(document).ready(function() {
    $("#refresh_dst").on("click", refresh);
});
```

Do lado do serviço é necessário implementar um método que devolva a distância. Só que neste caso a solução é mais complexa porque o método tem que invocar uma função para calcular a distância. Para esse efeito começe por acrescentar no início da aplicação **app.py** os pacotes **math** e **json** e a declaração da função **distance** para calcular a distância. E de seguida acrescente também à classe **Root** o método **dst**.

```
...
from math import radians, cos, sin, asin, sqrt
import json

def distance(lat, lon):
    lat1 = 38.752667
    lon1 = -9.184711
    lon, lat, lon1, lat1 = map(radians, [lon, lat, lon1, lat1]) # Graus -> rads
    dlon = lon - lon1
    dlat = lat - lat1
    a = sin(dlat/2)**2 + cos(lat1) * cos(lat) * sin(dlon/2)**2 # Haversine
    c = 2 * asin(sqrt(a))
    km = 6367 * c # 6367=raio da terra

    cherrypy.response.headers["Content-Type"] = "application/json"
    return json.dumps({"distance": km})
...

class Root:
    ...

    # Distância
    @cherrypy.expose
    def dst(self, lat, lon):
        cherrypy.response.headers["Content-Type"] = "application/json"
        return distance(float(lat), float(lon)).encode("utf-8")
```

Exercício 5.16

Faça as alterações sugeridas. Relance a aplicação e teste esta nova funcionalidade acionando o botão de *refresh*.

5.6 Acesso a imagens

A aplicação mostra na página de entrada (Tópico C - Imagem) uma imagem usando para esse efeito a marca **`img`**. Seria interessante poder aceder a imagens através do navegador, visualizá-las e armazená-las na nossa aplicação. Para esse efeito vamos usar a página **`upload.html`** acessível no Tópico D - UpLoad. Primeiro é preciso um elemento **`<input>`** especificando que se pretende aceder a imagens. Para poder visualizar a imagem é preciso usar uma função **`JavaScript`** que depois da imagem ter sido seleccionada a vai apresentar no local da página pretendido. Neste caso é necessário usar um elemento **`<canvas>`**.

Este elemento **`<canvas>`** é semelhante ao elemento **`<img>`**, com a diferença que é possível desenhar para ele em tempo real, enquanto que um elemento **`<img>`** apresenta uma imagem estática.

Estes dois elementos HTML são adicionados no elemento de classe **`content_image`** (marca **`<div>`**), logo no início da página **`upload.html`** a seguir ao cabeçalho já existente (**`<h1>`**).

```
...
<div class="content_image">
    <div class="btn btn-primary btn-block btn-input">
        <input type="file" accept="image/*" onchange="updatePhoto(event)"></input>
        <canvas id="photo" width="530" height="400">
    </div>
</div>
...
```

Para processar a imagem (visualizar e armazenar) precisamos de utilizar **`JavaScript`**. Crie um ficheiro designado por **`imagem.js`** com o código que se apresenta de seguida. E não se esqueça de o incluir no cabeçalho da página **`upload.html`**.

A função **`updatePhoto()`** vai apresentar a imagem selecionada. O pedido é activado quando o elemento **`<input>`** receber um evento, ou seja, quando se clicar no botão primário associado à entrada de imagens. A função irá apresentar a imagem selecionada no elemento **`<canvas>`** com a dimensão indicada (530×400). Por sua vez a função **`sendFile()`** vai armazenar a imagem. Ela recorre ao método **`POST`** para enviar para o servidor o nome do ficheiro que deve ser armazenado. Existe ainda a função **`updateProgress()`** que assim que a imagem estiver armazenada aciona um *alert* para que o utilizador saiba que o armazenamento da imagem está concluído.

```
function updatePhoto(event) {
    var reader = new FileReader();
    reader.onload = function(event) {
        //Criar uma imagem
        var img = new Image();
        img.onload = function() {
            //Colocar a imagem no ecrã
            canvas = document.getElementById("photo");
            ctx = canvas.getContext("2d");
            ctx.drawImage(img,0,0,img.width,img.height,0,0,530, 400);
        }
        img.src = event.target.result;
    }

    //Obter o ficheiro
    reader.readAsDataURL(event.target.files[0]);
    sendFile(event.target.files[0]);

    //Libertar recursos da imagem selecionada
    windowURL.revokeObjectURL(picURL);
}

function sendFile(file) {
    var data = new FormData();
    data.append("myFile", file);
    var xhr = new XMLHttpRequest();
    xhr.open("POST", "/upload");
    xhr.upload.addEventListener("progress", updateProgress, false);
    xhr.send(data);
}

function updateProgress(evt){
    if(evt.loaded == evt.total) alert("Okay");
}
```

Do lado do servidor é necessário receber o ficheiro e armazená-lo, ou seja copiar a informação para um local mais permanente, usando para isso o método **upload**. Isto porque os dados serão apagados na última linha da função **updatePhoto()**.

Podíamos armazená-las no diretório **images**. Mas vamos utilizar outro diretório, designado por **uploads**. Vamos portanto separar as imagens estáticas que ornamentam as páginas (HTML) das que são carregadas dinamicamente pela aplicação.

Por isso é muito importante antes de mais criar o diretório **uploads**. Senão não será possível guardar as imagens no servidor. E depois também é preciso acrescentar ao dicionário **config** uma nova entrada que é em tudo semelhante ao **images** excepto no nome, para criar o caminho para este novo diretório. Finalmente é preciso acrescentar à aplicação **app.py** o método **upload** na classe **Root** com o seguinte código.

```
...


class Root:
    ...

    # UpLoad
    @cherrypy.expose
    def upload(self, myFile):
        fo = open("uploads/" + myFile.filename, "wb")
        while True:
            data = myFile.file.read(8192)
            if not data: break
            fo.write(data)
        fo.close()
```

Exercício 5.17

Faça as alterações sugeridas. Relance a aplicação e teste esta nova funcionalidade selecionando várias imagens na página **UpLoad**. Não se esqueça de consultar o diretório **uploads** e verificar que as imagens estão mesmo a ser armazenadas.

A questão que se coloca agora é como podemos acrescentar persistência à nossa aplicação. Ou seja como é que, para além de carregar imagens podemos também visualizá-las através da própria aplicação. Para isso precisamos de:

- Uma página HTML para apresentar as imagens.
- Código **JavaScript** para mostrar as imagens nessa página.
- Um método **GET** para pedir ao servidor a informação sobre as imagens armazenadas.
- Uma base de dados com pelo menos uma tabela com a informação relevante acerca de cada imagem, nomeadamente o seu nome. Uma vez que a sua localização está perfeitamente definida.
- Acrescentar ao método **upload** já existente no servidor as instruções necessárias para atualizar a base de dados.
- Um novo método no servidor para consultar a base de dados e devolver a informação relativa às imagens armazenadas.

Glossário

API	Application Programming Interface
CERN	Conseil Européen pour la Recherche Nucléaire
CGI	Common Gateway Interface
CSS	Cascading Style Sheets
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
JS	JavaScript
JSON	JavaScript Object Notation
URL	Uniform Resource Locator
URI	Uniform Resource Identifier
W3C	World Wide Web Consortium
WWW	World Wide Web
WSGI	Web Server Gateway Interface
XML	Extensible Markup Language

Referências

[1] E. T. Bray, *The JavaScript Object Notation (JSON) Data Interchange Format*, RFC 7159, Internet Engineering Task Force, mar. de 2014.

[2] W3C. (1999). «HTML 4.01 Specification», URL: `http://www.w3.org/TR/1999/REC-html401-19991224/`.

[3] M. Mealling e R. Denenberg, *Report from the Joint W3C/IETF URI Planning Interest Group: Uniform Resource Identifiers (URIs), URLs, and Uniform Resource Names (URNs): Clarifications and Recommendations*, RFC 3305 (Informational), Internet Engineering Task Force, ago. de 2002.

[4] W3C. (2001). «Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification», URL: `http://www.w3.org/TR/2011/REC-CSS2-20110607/`.

[5] ECMA International, *Standard ECMA-262 – ECMAScript Language Specification*, Padrão, dez. de 1999. URL: `http://www.ecma-international.org/publications/standards/Ecma-262.htm`.