Tutorial 1

Servidor WEB com Conteúdos Estáticos e Dinâmicos

Objetivos:

- Criar um Servidor WEB com Conteúdos Dinâmicos sem Persistência

1.1 Introdução

Como já foi referido anteriormente, um servidor *Web* é um programa de computador que comunica um recurso (página HTML, imagem, vídeo, audio) a um cliente (*Web browser*) através do protocolo HTTP.

Um servidor *Web* é dinâmico se permitir aos utilizadores requisitar serviços como, por exemplo, aceder a imagens ou a dados que não foram previamente carregados no servidor quando este foi desenvolvido. E diz-se que tem persistência (também designado por memória) se, por exemplo, manter a informação sobre as imagens carregadas pelo servidor de modo a ser possível apresentá-las posteriormente, o que implica armazenar a sua localização e dados relevantes sobre elas numa base de dados.

Vamos começar por criar um servidor *Web* dinâmico simples sem persistência. Para esse efeito vamos utilizar o servidor *Web* desenvolvido na disciplina de Introdução à Engenharia Informática no guião de Integração de componentes em páginas *Web*.

1.2 Servidor *Web* sem persistência

Um servidor *Web* dinâmico sem persistência é normalmente organizada tal como se apresenta na Figura 1.1.

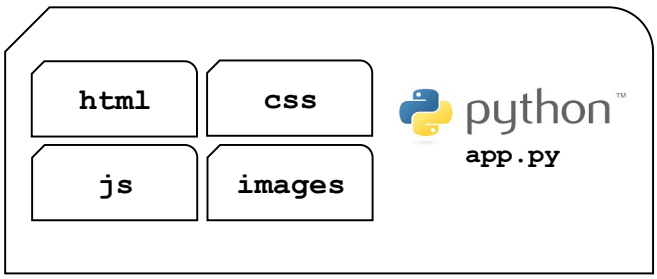

Figura 1.1: Servidor Web sem persistência.

Assim temos que ele é composto pelos seguintes componentes:

- Um programa desenvolvido na linguagem de programação *Python* designado por **app.py** para permitir requisitar conteúdos dinâmicos;
- O diretório **html** onde se encontram as páginas HTML que definem a estrutura do servidor, entre elas a página de arranque que habitualmente se designa por **index.html**;
- O diretório **css** para os ficheiros CSS de estilos pessoais que configuram o aspeto das página e de *frameworks* de estilos que permitem acelerar o seu desenvolvimento;
- O diretório **js** para os ficheiros de JS que dão dinâmica às páginas;
- O diretório **images** para as imagens estáticas (figuras e ícones) que ilustram as páginas.

1.2.1 Definição dos diretórios estáticos

Para ter acesso aos diversos recursos do servidor é conveniente definir os caminhos para os diretórios (**html**, **css**, **js** e **images**) e assim servir automaticamente os ficheiros constituintes do servidor.

O *CherryPy* permite definir uma variável de configuração (normalmente designada por **config**), que a partir do diretório onde o servidor é executado, designado por **baseDir**, cria um dicionário com os caminhos para todos os diretórios constituintes do servidor, tal como se apresenta de seguida e que constitui a estrutura básica do programa **app.py**.

O servidor expõe o método **index**, na raiz do URL (/), que é responsável por apresentar a página inicial do servidor Web, a página **index.html** que se encontra no diretório **html**. É a partir dela que todas as restantes páginas HTML poderão ser acedidas pelo utilizador do servidor. **Mais nenhuma página HTML precisa ou deve ser exposta.**

```python
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)
```

Exercício 1.1

No sítio da disciplina existe um arquivo ZIP chamado **`TutorialWEB1.zip`**. Obtenha este ficheiro, coloque-o dentro do diretório da disciplina e extraia o seu conteúdo.

Entre no diretório e compare o seu conteúdo (diretórios e ficheiros) com a Figura 1.1. Veja os ficheiros existentes em cada diretório.

De seguida execute o servidor Web (**`python3 app.py`**).

No navegador *Web* aceda ao servidor através do endereço `http://localhost:8080/`.

Navege no servidor 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 **`about.html`**. Aceda ainda ao Topic D - UpLoad e verifique que a página **`upload.html`** ainda está incompleta uma vez que não faz nada.

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

1.2.2 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 (**`Clock Refresh`**).
- Código *JavaScript* que obtenha a informação do servidor.
- Um método 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 **`<div>`**), colocando o seguinte excerto de código no ínicio do Topic A, logo a seguir ao comentário **`<!- Clock ->`**.

```
<!-- Clock -->
<div class="content_clock">
    <button id="refresh_clock" class="btn btn-block btn-primary">Clock Refresh</button>
        <div style="border: 2px solid black;">
            <div id="clock" style="width:100%; 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`**, a seguir ao respetivo comentário.

```
function clock() {
    $.get("/clock",
          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** do servidor o método **clock** com o seguinte código. Tem também que importar o módulo **time** no início do programa **app.py**.

```
...
import time
...

class Root():
    ...

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

Exercício 1.2

Faça as alterações sugeridas. Relance ao servidor 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_dist**, colocando o seguinte excerto de código no Topic A, logo a seguir ao comentário **<!- Distance ->**.

```
<!-- Distance -->
<div class="content_dist">
    <button id="refresh_dist" class="btn btn-block btn-primary">
        Distance to Stadium Refresh
    </button>
    <div style="border: 2px solid black;">
        <div id="distance" style="width:100%; text-align:center;"></div>
    </div>
</div>
```

Também precisa de criar um ficheiro designado por **distance.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 **distance**.

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

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

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

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

```
...
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]) # Degrees -> Radians
    dlon = lon - lon1
    dlat = lat - lat1
    a = sin(dlat/2)**2 + cos(lat1) * cos(lat) * sin(dlon/2)**2 # Haversine formula
    c = 2 * asin(sqrt(a))
    km = 6367 * c # Earth Ray = 6367 km

    cherrypy.response.headers["Content-Type"] = "application/json"
    return json.dumps({"distance": km}).encode("utf-8")
...

class Root():
    ...

    # Distance
    @cherrypy.expose
    def distance(self, lat, lon):
        return distance(float(lat), float(lon))
```

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 do programa **app.py** os módulos **math** e **json** e a declaração da função **distance** para calcular a distância. E de seguida acrescente também à classe **Root** o método **distance**.

Exercício 1.3

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

1.3 Acesso a imagens

O servidor mostra na página de entrada (Topic C - Image) uma imagem usando para esse efeito a marca **img**. Seria interessante poder aceder a imagens através do navegador, visualizá-las e armazená-las no nosso servidor.

Para esse efeito vamos usar a página **upload.html** acessível no Topic 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 ** **), 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)">
        <canvas id="photo" width="530" height="400"></canvas>
    </div>
</div>
...
```

Para processar a imagem (visualizar e armazenar) precisamos de utilizar **JavaScript**. Crie um ficheiro designado por **image.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 o evento **onchange**, ou seja, quando se clica no botão 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. Repare que esta função é mais sofisticada que as funções anteriormente desenvolvidas e utiliza o método **FormData()**. Este método fornece uma maneira de criar um conjunto de pares chave/valor (dicionário) representando campos de formulário e seus valores, que podem ser enviados usando o método **XMLHttpRequest()**.

Este método por sua vez usa o método HTTP **POST** para enviar para o servidor o ficheiro que deve ser armazenado. Utiliza também o método **addEventListener()** para determinar quando o armazenamento da imagem está concluída. Recorrendo para esse efeito ao evento **progress** e à função **updateProgress()**, que assim que a imagem estiver completamente armazenada aciona um *alert* para informar o utilizador.

```
function updatePhoto(event) {
    var reader = new FileReader();
    reader.onload = function(event) {
        // Create an image
        var img = new Image();
        img.onload = function() {
            // Show the image on the screen
            const canvas = document.getElementById("photo");
            // Using jQuery it is const canvas = $("#photo")[0];
            const ctx = canvas.getContext("2d");
            ctx.drawImage(img,0,0,img.width,img.height,0,0,530,400);
        }
        img.src = event.target.result;
    }

    // Get the file
    reader.readAsDataURL(event.target.files[0]);
    sendFile(event.target.files[0]);

    // Free image resources
    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(this), 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 pelo servidor.

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 ao servidor **app.py** o método **upload** na classe **Root** com o seguinte código.

```
...
class Root():
    ...

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

Exercício 1.4

Faça as alterações sugeridas. Relance o servidor 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.

Neste sistema de armazenamento dos ficheiros de imagens, em que os ficheiros são registados com os seus nomes originais no mesmo diretório, vamos inevitavelmente ter um problema. Se dois ou mais utilizadores carregarem no servidor ficheiros de imagens com o mesmo nome, eles vão sendo substituídos e perdidos. Por isso precisamos de usar um outro nome que não o nome original do ficheiro. Uma alternativa consiste em usar um nome criado a partir de uma síntese do conteúdo da imagem.

Exercício 1.5

Altere o código do método **UpLoad** de maneira a guardar os ficheiros com um nome dado pela síntese do conteúdo da imagem. Use, por exemplo, a função de síntese **sha256** da biblioteca **hashlib**. Tenha em atenção que deve manter a mesma extensão do ficheiro, porque ela define o tipo de ficheiro de imagem.

Glossário

CSS	Cascading Style Sheets
HTML	HyperText Markup Language
JS	JavaScript
JSON	JavaScript Object Notation
URL	Uniform Resource Locator