Tema 17

Aplicações e Serviços Web

Objetivos:

- Servidores Web
- Serviços Web

17.1 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.

17.2 Servidores *Web*

17.2.1 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 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.

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*). 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.

O *CherryPy* é composto por 8 módulos:

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

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

`CherryPy.tools` Conjunto de ferramentas ortogonais 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* Common Gateway Interface (CGI) seguinte. Revendo cada linha do exercício seguinte, 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. 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 17.1

Crie no seu próprio computador o seguinte ficheiro.

```
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/`.

Exercício 17.2

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.

Exercício 17.3

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 "You 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.

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 17.4

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**).

17.2.2 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 17.5

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 17.6

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".

17.3 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 17.7

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 17.8

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 17.9

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

17.4 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 necessário.

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 17.10

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 17.11

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.

17.5 Para Aprofundar

Exercício 17.12

Recupere o exercício de aprofundamento do guião anterior.

Construa uma aplicação *Web* que aceda ao ficheiro disponível em `http://www.ipma.pt/resources.www/internal.user/pw_hh_pt.xml`, contendo os dados meteorológicos observados nas principais cidades portuguesas.

A sua aplicação deverá receber o nome da cidade por método `POST`, pelo que deve construir um formulário com a lista de cidades possíveis usando uma *dropbox*.

À submissão do formulário deverá seguir-se a impressão dos dados da cidade indicada no formulário.

Para acesso ao serviço será necessário adicionar um cabeçalho (header) ao pedido:

```
...
url = "http://www.ipma.pt/resources.www/internal.user/pw_hh_pt.xml"
requests.get(url, headers={"referer": "http://www.ipma.pt/pt/index.html"})
...
```

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`.