Laboratórios de Informática 2016–2017

DETI - Universidade de Aveiro

Créditos Estes guiões constituem um trabalho continuado de refinamento que contou com a colaboração de João Paulo Barraca, Diogo Gomes, André Zúquete, João Manuel Rodrigues, António Adrego da Rocha, Tomás Oliveira e Silva, Sílvia Rodrigues e Óscar Pereira.

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 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 17.1

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 17.2

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]

[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

17.2.2 Servidores Aplicacionais

Na secção anterior abordámos a interface CGI que se popularizou nos finais do século passado como a ferramenta para desenvolver conteúdos dinâmicos para a *Web*. Servidores como o *Apache* e o *Microsoft IIS* permitem a execução de programas externos usando a interface CGI e ainda hoje diversos sites *Web* fazem uso desta tecnologia. No entanto, o crescente dinamismo da *Web* e a necessidade de criação de aplicações *Web* que requerem múltiplas interações com o utilizador tornam a interface CGI extremamente ineficiente e até mesmo insegura (já que os programas correm com as mesmas permissões do servidor *Web*).

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 executar um programa externo, mas na de o próprio programa incluir o servidor *Web*.

Neste capítulo, vamos abordar um servidor aplicacional específico para *Python*. O *CherryPy* é um servidor aplicacional maduro usado tanto para pequenas aplicações como para grandes (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 python-cherrypy3
```

Em alternativa pode executar:

```
sudo pip install CherryPy
```

ou:

```
sudo pip install --upgrade CherryPy
```

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

argumentos.

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* CGI anterior.

Exercício 17.3

Crie no seu próprio computador o seguinte ficheiro.

```
import cherrypy

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

cherrypy.tree.mount(HelloWorld(), "/")
cherrypy.server.start()
```

Se possuir a última versão do *CherryPy* as duas linhas finais do ficheiro anterior podem ser substituídas por:

```
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/`.

Revendo cada linha do exercício anterior, 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 17.4

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

anteriormente.

Exercício 17.5

Crie um novo programa com o seguinte conteúdo

```
import cherrypy

class Node(object):
    @cherrypy.expose
    def index(self):
        return "Eu sou um objecto Folha"

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

    @cherrypy.expose
    def index(self):
        return "Eu sou o objecto Raiz"

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

if __name__ == "__main__":
    cherrypy.tree.mount(Root(), "/")
    cherrypy.server.start()
```

Aceda a cada um dos recursos a partir do seu navegador *Web* (**/page**, **/node/**).

Exercício 17.6

Acrescente agora uma nova classe **HTMLDocument** que devolva o conteúdo de um ficheiro HTML lido do disco.

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.

17.2.3 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 17.7

Crie uma página HTML com o código para formulário 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>
```

Não esquecer de completar a página com o código HTML apropriado.
Crie um novo método na sua aplicação:

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

No exercício anterior permitimos ao nosso servidor aplicacional servir uma página HTML com o conteúdo de um formulário usando o método **`serve_file`**. No entanto, a submissão

do formulário de *login* necessita ainda da implementação de mais um método.

Exercício 17.8

Crie um novo objeto (*actions*) e método na sua aplicação. Não se esqueça de associar o novo objeto à Raiz.

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

Abra o formulário através do endereço `http://localhost:8080/form/`, preencha-o e submeta.

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

17.3 Serviços Web

Na secção anterior vimos como um cliente *Web* pode interagir com uma aplicação *Web* alojada no servidor. Nesta secção vamos ver 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 **`urllib2`**, cuja documentação completa encontra-se disponível em `http://docs.python.org/2/library/urllib2.html`.

A biblioteca **`urllib2`** permite-nos aceder a uma página *Web* de forma muito semelhante à que utilizamos em *Python* para aceder a um ficheiro.

```
import urllib2

f = urllib2.urlopen("http://www.python.org")
```

Exercício 17.9

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.

O uso directo do método **urlopen** 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.

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 *Python* possui na biblioteca **urllib** o método **urlencode** que permite converter um dicionário *Python* numa *String* codificada em **application/x-www-form-urlencoded**.

```
import urllib

data = urllib.urlencode({"nome": "Ana", "idade": 20})
```

Munidos dessa *String* codificada podemos construir um objecto **Request** que é usado como argumento de **urlopen**.

```
req = urllib2.Request(url)
req.add_data(data)

f = urllib2.urlopen(req)
```

Exercício 17.10

Fazendo uso da aplicação *Web* desenvolvida anteriormente 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 17.11

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

```
serviceurl = "http://maps.googleapis.com/maps/api/geocode/json?"

url = serviceurl + urllib.urlencode({"sensor":"false", "address": address})
f = urllib.urlopen(url)
```

17.4 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.

Glossário

API Application Programming Interface

CERN Conseil Européen pour la Recherche Nucléaire

CGI Common Gateway Interface

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, endereço: `http://www.w3.org/TR/1999/REC-html401-19991224/`.