Desenvolvimento de Aplicações Web Documentation

Versão 0.1

Eduardo Klosowski

set 27, 2017

Conteúdo

Este texto mostrará como é a interação das linguagens de programação com o servidor, bem como algumas possibilidades de estrutura para as aplicações Web, envolvendo a programação no servidor e cliente, de forma independente da linguagem de programação.

Introdução

Requisitos

Os exercício foram feitos utilizando a linguagem Python. Esta linguagem foi a escolhida por ser amigável com iniciantes e já vir instalada em diversos sistemas, como GNU/Linux e Mac OS.

Os códigos foram testados para funcionar nas versão 3.4 ou mais recentes do Python, mas podem ser utilizados em versões anteriores como a 2.7 com os ajustes necessários. Em sistemas que possuem o Python 2 como padrão, porém possuem o Python 3 disponível, troque o comando `python` por `python3`.

É recomendável ter instalado o pip no sistema para a instalação de pacotes do Python, ou verifique no gerenciamento de pacotes da sua distribuição por cada um dos pacotes quando necessário.

Licença

Capítulo 01 - A Web

Para o completo funcionamento de uma aplicação Web, que não deixa de ser um site, existe a interação de diversos elementos. Alguns desses elementos são conhecidos por todos, como o navegador (também conhecido como *browser* ou *web browser*), outros ficam mais escondidos, como o servidor. Estes dois elementos já formam um modelo mínimo, baseado no modelo cliente-servidor, onde o navegador tem o papel de cliente, acessando o servidor.

Explicando este modelo de forma mais detalhada, o código da aplicação Web fica no servidor, que quando requisitado pelo cliente, neste caso o navegador, permite o acesso a aplicação a este cliente. Vale observar que o navegador e o servidor são dois programas diferentes, podendo estar no mesmo computador em alguns casos, mas na internet o

mais comum são computadores diferentes, comunicando via rede, podendo ter sistemas operacionais e até arquiteturas completamente diferentes. Essa diversidade é possível devido a Web seguir padrões, e qualquer programa que implemente estes padrões podem ser utilizados.

Navegador

Um navegador conecta a um servidor e faz requisições, por isso é chamado de cliente no modelo cliente-servidor. Para que uma página seja acessada, o navegador conecta no servidor correspondente, requisitando o arquivo da página para que a mesma possa ser exibida.

Existem diversos navegadores, alguns funcionam através de interfaces gráfica, outros através de interfaces de texto, ou ainda via linha de comando. Dependendo do contexto de aplicação, um tipo pode ser mais adequado que outro, também é possível ter vários navegadores num computador e executá-los simultaneamente. Alguns exemplos de navegadores são:

- Gráficos:
  - Mozilla Firefox
  - Google Chrome
  - Internet Explorer
- Texto:
  - links2
  - lynx
  - w3m
- Linha de comando:
  - wget
  - curl
  - HTTPie

Servidor

Um servidor é um programa que fica escutando numa porta específica, quando um cliente se conecta e faz alguma requisição, a mesma é processada e uma resposta é devolvida ao cliente. Uma forma fácil de se entender este processo é imaginar arquivos. Num servidor imaginário existem dois arquivos `a.png` e `b.png`, quando o cliente conecta e requisita `a.png`, o servidor pega este arquivo e o envia via rede ao cliente, o mesmo ocorre com `b.png`.

Existem uma infinidade de servidores, cada um com características próprias, como exigir menos memória, responder uma maior quantidade de requisições simultaneamente, suporte a diversas tecnologias e assim por diante. Exemplo dos servidores mais conhecidos e utilizados são:

- Apache
- NGINX

- lighttpd
- IIS

Também é possível ter mais de um servidor no mesmo computador, porém a combinação de IP e porta de escuta tem que ser única para cada servidor.

URL

Praticamente tudo na Web pode ser acessada via uma URL. Ela possui vários campos que descrevem a localização de um determinado recurso. Seu formato pode ser descrito como:

```
protocolo://usuário:senha@servidor:porta/recurso
```

- **Protocolo**: normalmente em aplicações Web é `http`, ou `https` quando utilizado criptografia.
- **Usuário** e **Senha**: podem ser fornecidos diretamente na URL quando uma autenticação se faz necessária para acessar determinado recurso. Esses campos são opcionais.
- **Servidor**: endereço IP ou nome DNS do servidor em qual o recurso encontra-se disponível.
- **Porta**: número da porta (TCP ou UDP, de acordo com o protocolo). Esse campo é opcional, caso não seja informado será utilizada a porta padrão do protocolo, `80` para HTTP, `443` para HTTPS e assim por diante. Uma relação completa pode ser vista no arquivo `/etc/services` em sistemas UNIX.
- **Recurso**: endereço do recurso que desejá-se acessar, também pode conter várias barras (`/`).

Exemplos de URL válidas:

`http://localhost/` Requisição do protocolo HTTP para o nome `localhost`, na porta `80`, sem autenticação, recurso `/`.

`https://127.0.0.1/` Requisição do protocolo HTTPS (criptografado) para o endereço IP `127.0.0.1`, na porta `443`.

`http://localhost:8000/` Requisição na porta `8000`.

`http://localhost:8000/arquivo.html` Requisição do recurso `/arquivo.html`.

`http://guest@localhost:8000/imagem/foto.jpg` Requisição com usuário `guest`, recurso `/imagem/foto.jpg`.

`http://admin:secreta@localhost:8000/segredo/` Requisição com usuário `admin` e senha `secreta`, recurso `/segredo/`.

> **Aviso: Cuidado com usuário e senha nas URL!**
>
> Qualquer pessoa que conseguir visualizar a URL terá acesso aos mesmos. Muitas vezes é preferível não colocá-las e digitar quando o acesso for feito, ou colocar apenas o usuário, pedindo apenas a senha para acessar.

A URL também pode ser utilizada para indicar onde encontra-se um serviço, como a baixo que indica as informações para acesso a um banco de dados PostgreSQL:

```
postgres://siteweb:senhamuitosecreta@db.sistema.com.br:5432/bancodedados
```

Alguns parâmetros opcionais também podem ser passados numa URL, seguindo um formato chave-valor. Para isto basta colocar o símbolo de interrogação (`?`) no final, seguindo do nome do parâmetro, símbolo de igual (=) e seu valor. Mais de um parâmetro pode ser informado utilizando o e comercial (`&`). Exemplos:

```
http://localhost/?pagina=home
http://localhost/login.html?username=admin&password=secreta
```

Nota: Qualquer caractere diferente de uma letra, número, traço (-) ou underscore (_) em algum campo da URL deve ser codificado!

Isso se faz necessário para não criar confução entre o valor de um campo com o formato da URL em sí. Normalmente se utiliza o símbolo de porcentagem seguido da representação em hexadecimal do caracter em unicode. Exemplo: `%20` é um espaço em branco e `%21` é uma exclamação (`!`).

Isto é mais utilizado em nome de arquivos, usuário, senha e parâmetros opcionais, nos demais campos é preferível evitar a utilização de caracteres especiais.

Um caso especial é o símbolo +, que também significa um espaço em branco, da mesma forma que o `%20`, e não o símbolo propriamente dito.

Nota: Toda vez que for acessar um servidor que está executando no próprio computador, pode ser utilizado os endereços `127.0.0.1` ou `localhost`, porém eles são nomes diferentes, tanque que o navegador guardará informações diferentes para cada um, além do servidor também poder tratar os acessos de forma diferente.

Caso queira utilizar algum outro nome, porém sem configurá-lo no servidor DNS, o mesmo pode ser feito no arquivo `/etc/hosts` em sistemas UNIX.

Protocolo HTTP

Conforme comentado anteriormente, para permitir que esses diferentes programas se comuniquem existe um padrão. Este padrão é chamado de protocolo HTTP, que descreve o formato das requisições enviadas pelos clientes, e das respostas enviadas pelos servidores.

Programas que enviam requisições, segundo o protocolo HTTP, também são chamados de clientes HTTP. Da mesma forma, os programas que responde as requisições, segundo o protocolo HTTP, também são chamados de servidor HTTP.

Como o protocolo HTTP é um protocolo de rede, ele se baseia em vários outros protocolos para o seu funcionamento, sendo os mais importantes:

- IP: atribui um endereço a cada dispositivo conectado na rede, permitindo a identificação para qual equipamento uma requisição deve ser enviada, ou de qual equipamento originou-a e aguarda resposta.
- DNS: atribui nomes mais fáceis de lembrar aos endereços numéricos do IP.
- TCP: define um processo para a comunicação entre programas diferentes, garantindo que a integridade das informações enviadas baseado em verificações.
- TLS/SSL: permite a criptografia da comunicação do protocolo HTTP, que quando utilizado também é conhecido por HTTPS.

O primeiro passo para ter uma aplicação Web publicada é configurar um servidor HTTP, o processo consiste em escutar uma porta TCP, que por padrão é a `80`. Depois que o servidor estiver em execução, o cliente poderá se conectar nesta porta, enviando suas requisições.

Requisição HTTP

O protocolo HTTP define o padrão que as requisições devem seguir. O padrão é em texto claro, separado em duas sessões principais cabeçalho e corpo. O cabeçalho define principalmente qual recurso que está sendo requisitado, junto com algumas informações do cliente. O corpo é uma parte que nem sempre está presente, que pode conter os dados de um formulário, ou um arquivo enviado ao servidor, por exemplo.

Um exemplo de uma requisição apenas com cabeçalho:

```
GET /home.html HTTP/1.1
Accept: */*
Accept-Encoding: gzip, deflate
Connection: keep-alive
Host: localhost:8000
User-Agent: HTTPie/0.9.2
```

Um exemplo de uma requisição com cabeçalho e corpo:

```
POST /login HTTP/1.1
Accept: */*
Accept-Encoding: gzip, deflate
Connection: keep-alive
Content-Length: 31
Content-Type: application/x-www-form-urlencoded; charset=utf-8
Host: localhost:8000
User-Agent: HTTPie/0.9.2

username=usuario&password=senha
```

No segundo exemplo pode-se notar que a divisão entre o cabeçalho e o corpo é feita com uma linha em branco, esta linha que define o fim do cabeçalho. Em ambos os casos a primeira linha indica o método da requisição (`GET` e `POST`), endereço do recurso (`/home.html` e `/login`), terminando com a versão do protocolo (`HTTP/1.1`). Nas demais linhas do cabeçalho existem algumas informações extras no formato de chave-valor, separado por dois pontos (`:`).

Para o desenvolvimento de uma aplicação não é necessário entender toda uma requisição profundamente, porém as ideias de método, endereço do recurso, cabeçalho e corpo são importantes.

No método existem algumas palavras chaves, sendo as principais `GET` e `POST`. Existe uma diferença na semântica dos dois, porém a principal diferença é que uma requisição `GET` não possui corpo, enquanto uma requisição `POST` possui.

Endereço de recurso pode ser entendido como o endereço ou caminho de um arquivo, ao se configurar um servidor para servir arquivos de um diretório via HTTP, isso é completamente válido. Porém não é limitado apenas a endereços de arquivos, como explicado mais adiante.

Algumas informações interessantes no cabeçalho destas requisições são:

- `Host`: indica que site foi acessado pelo cliente, bastante importante para quando existem mais de um site disponível no mesmo servidor, ou quando o acesso é feito via proxy.
- `User-Agent`: uma assinatura do programa que fez a requisição, aqui pode-se ver que foi utilizado o HTTPie, porém esta informação não é confiável, uma vez que pode ser alterada facilmente.
- `Content-Type` (apenas no `POST`): formato utilizado no corpo da requisição.

Com relação ao corpo da requisição `POST`, está codificada no formato `application/x-www-form-urlencoded`, utilizando `utf-8` para a representação dos caracteres, conforme descrito em `Content-Type`. Este formato é utilizado principalmente para formulários, e tem o seu funcionamento conforme descrito nos parâmetros opcionais da URL.

Resposta HTTP

Depois que uma linha em branco demarca o fim do cabeçalho da requisição HTTP, o seu corpo foi recebido pelo servidor se estiver presente na requisição, a mesma será processada e devolvida na forma de uma resposta HTTP. A resposta também se divide em cabeçalho e corpo, assim como a requisição, porém possui algumas diferenças:

```
HTTP/1.0 200 OK
Content-Length: 15
Content-type: text/html
Date: Thu, 24 Dec 2015 01:33:41 GMT
Last-Modified: Thu, 24 Dec 2015 01:33:37 GMT
Server: SimpleHTTP/0.6 Python/3.4.2

<h1>Teste</h1>
```

Na primeira linha tem a versão do protocolo, pode-se notar que este servidor não suporta completamente a versão 1.1, e respondeu conforme a versão 1.0. Logo em seguida tem o código da resposta e o mensagem, neste caso `200 OK`. Existem algumas famílias de código de resposta, sendo as mais comuns descritas a baixo:

Família	Descrição	Exemplo
2XX	Requisição recebida e processada corretamente.	`200 OK`
3XX	Redirecionamentos, quando o recurso encontra-se em outro lugar.	`301 Moved permanently`
4XX	O recurso não pode ser acessado pelo cliente.	`404 File not found`
5XX	Erro durante o processamento da requisição no servidor.	`500 Internal server error`

Algumas coisas que pode-se notar neste cabeçalho são:

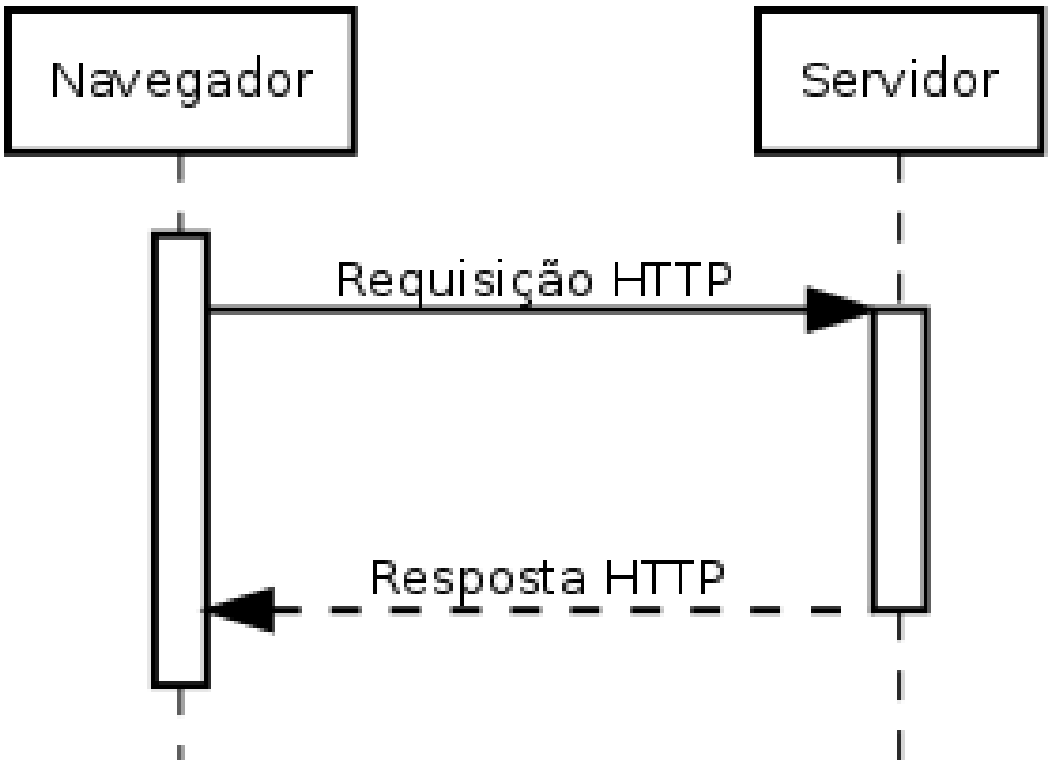
- **Content-Length**: indica o tamanho de corpo da resposta, útil em *downloads*, para se fazer uma previsão do término, quando não informado não é possível determinar quando o mesmo terminará.
- **Contnet-Type**: indica de qual é o formato do corpo da resposta, funciona da mesma forma que a requisição.
- **Date** e **Last-Modified**: indica o horário do servidor quando respondeu e a última modificação do arquivo, pode-se ser utilizado para fazer *cache* de uma página e não precisar baixá-la novamente caso não tenha sofrido nenhuma modificação.
- **Server**: uma assinatura do programa que fez a resposta, deve-se tomar cuidado na configuração do servidor em produção para não expor informações de mais que permitam identificar uma falha de segurança, como versões.

Logo em seguida tem o corpo da resposta que é a página propriamente dita.

Fluxo de requisição/resposta

Tudo no protocolo HTTP funciona baseado neste fluxo de enviar uma requisição e obter a resposta. Com isso é possível dizer que o HTTP não guarda estado, já que cada requisição é feita separadamente e uma requisição não tem relação direta com outra.

Porém existe uma forma de adicionar identificadores para saber quem é determinado cliente ou se o mesmo fez o login dentro da página. Isso é feito baseado em cookies, que é um campo no cabeçalho. Numa resposta o campo `Set-Cookie: nome=valor` define um cookie, na próxima requisição essa informação será passada no cabeçalho como `Cookie: nome=valor`.



Aviso: O valor dos cookies são enviados em texto claro sem criptografia, por este motivo evite guardar informações como senha. Prefira manter essa informação no servidor, passando apenas um código identificador da mesma, como um UUID, porém este código deve ser gerado aleatoriamente para não ser previsivél ou facilmente descoberto por terceiros.

Muitas linguagens ou frameworks implementam um recurso semelhante ao descrito sobre o nome de sessão.

Mimetype

Sempre que existe um corpo numa requisição ou resposta, é necessário dizer qual o tipo desta informação, como se fosse um formato. Esses tipos seguem o padrão mimetype, que é composto por duas partes separadas por barra (`/`), na primeira tem uma categoria e o formato na segunda. Essa informação é importante para que o cliente saiba o que fazer com esta informação, como qual o formato que deverá exibí-la.

Alguns exemplos de mimetypes são:

```
text/plain
text/html
application/json
image/png
video/webm
```

Em sistemas UNIX os mimetypes podem ser verificados no arquivo `/etc/mime.types` ou com o comando `mimetype <arquivo>`.

Algumas vezes também pose conter alguma informação adicional como `text/html; charset=utf-8`, que além de dizer que o arquivo é um HTML, já identifica qual o conjunto de caracteres que deve ser utilizado para ler seu conteúdo.

Exercícios

1. Crie um diretório no seu computador, com o arquivo `index.html` com o código a baixo:

```
<h1>Teste</h1>
```

2. Execute o comando a seguir dentro deste diretório.

```
python -m http.server
```

3. *(Opcional)* Faça requisições utilizando o HTTPie, alguns exemplos:

```
http http://localhost:8000/ -v
http http://localhost:8000/index.html -v
http http://localhost:8000/naoexiste.html -v
```

Nota: É possível instalar o HTTPie com o comando `pip`. Exemplo:

```
pip install httpie
```

---

4. Faça requisições utilizando um navegador gráfico como o Mozilla Firefox:

```
http://localhost/
http://localhost:8000/
http://127.0.0.1:8000/
http://localhost:8000/index.html
```

5. Execute o comando a baixo e acesse `http://localhost:8080/`:

```
python -m http.server 8080
```

5. *(Opcional)* Copie uma imagem para este diretório e tente abri-la no navegador.
6. *(Opcional)* Crie subdiretórios com arquivos e tente acessá-los.
7. *(Opcional)* Num diretório com vários arquivos, porém sem um `index.html`, execute um servidor HTTP e seja a página que será gerada automaticamente.

Discussão

1. Quando executado dois servidores ao mesmo tempo aparece o seguinte erro:

```
Traceback (most recent call last):
  File "/usr/lib/python3.4/runpy.py", line 170, in _run_module_as_main
    "__main__", mod_spec)
  File "/usr/lib/python3.4/runpy.py", line 85, in _run_code
    exec(code, run_globals)
  File "/usr/lib/python3.4/http/server.py", line 1241, in <module>
    test(HandlerClass=handler_class, port=args.port, bind=args.bind)
  File "/usr/lib/python3.4/http/server.py", line 1214, in test
    httpd = ServerClass(server_address, HandlerClass)
  File "/usr/lib/python3.4/socketserver.py", line 429, in __init__
    self.server_bind()
  File "/usr/lib/python3.4/http/server.py", line 133, in server_bind
    socketserver.TCPServer.server_bind(self)
```

```
  File "/usr/lib/python3.4/socketserver.py", line 440, in server_bind
    self.socket.bind(self.server_address)
OSError: [Errno 98] Address already in use
```

Por quê isso ocorre? Como é possível executar dois servidores simultaneamente?

Como é possível que um servidor responda com uma página para uma interface de rede, e com outra página para outra interface de rede?

Para que serve o comando `netstat` e no que ele pode ser útil?

2. Quando executado o comando a baixo é possível acessar os arquivos de outro computador.

```
python -m http.server
```

Por quê o mesmo não ocorre com o comando a baixo?

```
python -m http.server --bind 127.0.0.1
```

3. Por quê com um usuário comum não consegue executar o comando a baixo?

```
python -m http.server 80
```

O que a seguinte mensagem quer dizer?

```
Traceback (most recent call last):
  File "/usr/lib/python3.4/runpy.py", line 170, in _run_module_as_main
    "__main__", mod_spec)
  File "/usr/lib/python3.4/runpy.py", line 85, in _run_code
    exec(code, run_globals)
  File "/usr/lib/python3.4/http/server.py", line 1241, in <module>
    test(HandlerClass=handler_class, port=args.port, bind=args.bind)
  File "/usr/lib/python3.4/http/server.py", line 1214, in test
    httpd = ServerClass(server_address, HandlerClass)
  File "/usr/lib/python3.4/socketserver.py", line 429, in __init__
    self.server_bind()
  File "/usr/lib/python3.4/http/server.py", line 133, in server_bind
    socketserver.TCPServer.server_bind(self)
  File "/usr/lib/python3.4/socketserver.py", line 440, in server_bind
    self.socket.bind(self.server_address)
PermissionError: [Errno 13] Permission denied
```

Capítulo 02 - Código executado no servidor

Entre a requisição e a resposta do HTTP muita coisa pode ocorrer. Além de buscar um arquivo em disco e enviá-lo pela rede, outros processos podem serem feitos, já que o protocolo define como é a requisição e resposta, porém não como uma resposta deve ser gerada, isso abre opções para criar páginas dinâmicas.

Uma página dinâmica é uma página que tem o seu conteúdo gerado quando uma requisição é feita. Esse processo é feito através de um programa que identifica as informações da requisição HTTP, baseado num algoritmo gera a resposta.

CGI

O primeiro método padronizado para gerar páginas dinâmicas e mais simples é o CGI. Seu funcionamento em vez de carregar um arquivo, executa o mesmo, passando as informações do cabeçalho como variáveis de ambiente e corpo na entrada padrão. O conteúdo da resposta é feita através da saída padrão, com as informações de cabeçalho, uma linha em branco e o corpo da resposta. Sendo o campo `Content-Type` obrigatório na resposta.

O CGI é independente de linguagem de programação, já que qualquer linguagem que permita controlar a entrada padrão, saída padrão e ler as variáveis de ambiente pode ser utilizada. Um exemplo em shell script pode ser visto a baixo:

```
#!/bin/bash
echo "Content-Type: text/html; charset=utf-8"
echo
echo "<h1>Computador: $HOSTNAME</h1>"
```

Existem vários servidores que suportam CGI, inclusive o servidor da biblioteca padrão do Python, quando chamado com o parâmetro `--cgi`. Porém sempre é bom verificar como o servidor em questão funciona, neste servidor as páginas tem que estar no diretório chamado `/cgi-bin/`, em outros o arquivo deve ter a extensão `.cgi`.

Salvando o código a cima com o nome de `index.cgi` dentro do diretório `cgi-bin`, conforme a árvore a baixo:

```
(servidor)
 ` cgi-bin/
   ` index.cgi
```

Executando o script diretamente, uma saída possível é a seguinte:

```
Content-Type: text/html; charset=utf-8

<h1>Computador: notebook</h1>
```

Porém quando iniciado o servidor HTTP e feita a requisição a saída é outra:

```
python -m http.server --cgi
http http://localhost:8000/cgi-bin/index.cgi
```

```
HTTP/1.0 200 Script output follows
Content-Type: text/html; charset=utf-8
Date: Sun, 27 Dec 2015 23:49:35 GMT
Server: SimpleHTTP/0.6 Python/3.4.2

<h1>Computador: notebook</h1>
```

Um exemplo de script acessando variáveis de ambiente:

```
#!/usr/bin/env python
import os

print('Content-Type: text/plain')
print()
for key in ('REQUEST_METHOD', 'SCRIPT_NAME', 'REMOTE_ADDR', 'HTTP_USER_AGENT'):
    print('%s => %s' % (key, os.environ.get(key)))
```

```
HTTP/1.0 200 Script output follows
Content-Type: text/plain
Date: Mon, 28 Dec 2015 00:03:42 GMT
```

```
Server: SimpleHTTP/0.6 Python/3.4.2

REQUEST_METHOD => GET
SCRIPT_NAME => /cgi-bin/index.cgi
REMOTE_ADDR => 127.0.0.1
HTTP_USER_AGENT => HTTPie/0.9.2
```

Em Python existem duas bibliotecas que auxiliam no desenvolvimento de scripts CGI, sendo cgi e cgitb.

Fast-CGI

Scripts CGI são simples e fáceis de implementar, porém sua utilização tem problemas com desempenho, uma vez que cada página acessada, gera um processo diferente que tem que ser executado. Visando a resolver esse problema existe o Fast-CGI, que manter um processo sempre em execução, quando uma requisição HTTP chega no servidor o mesmo comunica com esse processo via socket num protocolo semelhante ao HTTP.

As vantagens deste método é que requisição HTTP não precisa necessariamente iniciar um processo novo. Também como a comunicação ocorre via socket, pode ser feita via unixsocket, quando no mesmo computador, ou via TCP, podendo estar em outro computador para dividir a carga. Porém sua implementação não é simples, uma vez que cria um protocolo próprio e precisa trabalhar com multiplexação de requisições.

Atualmente essa é uma alternativa que vem competindo com o `mod_php` do Apache. Juntando o NGINX que não tem suporte ao PHP por padrão, mas implementa o Fast-CGI, com o PHP FPM, entrega uma alternativa interessante de servidor.

WSGI

Python por sua vez criou um padrão próprio chamado WSGI, que é definido pela PEP 333, sendo mais tarde substituída pela PEP 3333. Sua principal característica é que o interpretador Python é executado dentro do servidor, sendo que as informações das requisições são passadas a uma única função, que processa e devolve a resposta.

A função do WSGI recebe dois parâmetros. No primeiro as informações da requisição são passadas, semelhante as variáveis de ambiente do CGI. No segundo uma função que recebe as informações do cabeçalho da resposta. O corpo da mesma é um interável que a função deve retornar.

Um exemplo de código WSGI, utilizando o servidor de referência da biblioteca padrão, pode ser visto a baixo:

```
1  def application(environ, start_response):
2      start_response('200 OK', [('Content-Type', 'text/html; charset=utf-8')])
3      yield '<h1>Teste<h1>'.encode('utf-8')
4
5
6  if __name__ == '__main__':
7      from wsgiref.simple_server import make_server
8
9      httpd = make_server('127.0.0.1', 8000, application)
10     print('Servidor rodando em http://127.0.0.1:8000/ ...')
11     httpd.serve_forever()
```

Quando a aplicação Web for colocada em produção, outro servidor deve ser utilizado. O gunicorn é um servidor HTTP simples que só funciona com WSGI, pode ser instalado com o `pip` e iniciado via linha de comando.

```
pip install gunicorn
gunicorn -b 127.0.0.1:8000 --access-logfile - --error-logfile - wsgi:application
```

Nota: O `wsgi:application` significa que a função WSGI se chama `application` e está no módulo `wsgi`, neste caso o arquivo `wsgi.py`.

O WSGI também pode ser utilizado semelhante ao Fast-CGI num servidor HTTP, a principal diferença é que o protocolo de comunicação é o HTTP e por isso deve ser configurado como um proxy reverso. Desta forma mantem as vantagens, como poder executar o código em outro computador, ou até configurar vários servidores para responder as solicitações, fazendo balanceamento de carga.

Middleware

Como o WSGI é uma função como qualquer outra em Python, é possível tirar proveito da flexibilidade da linguagem para implementar algumas funcionalidades. Middleware são funções que substitui a função da aplicação, podendo executar algo antes ou depois da aplicação, como verificar os cookies e adicionar no `environ` qual o usuário que está autenticado ou gravar o tempo que as páginas levam para serem geradas, fazendo uma análise de desempenho.

Como quase todos os frameworks Web de Python são baseados em WSGI, esses middlewares podem ser utilizados em conjunto com qualquer um desses frameworks.

Como programar para Web na linguagem ou framework X?

Esses métodos de gerar páginas dinâmica são apenas alguns exemplos, porém quando se programa numa linguagem ou framework isso é abstraido. No PHP o código que executa como CGI, Fast-CGI ou no Apache com o `mod_php` é o mesmo, diferenciando apenas o interpretador da linguagem. No Python quase todos são baseados em WSGI, podendo a partir dele criar alternativas para outros métodos, porém mesmo assim não é comum a manipulação da função WSGI diretamente.

Não importa qual a linguagem ou framework utilizado para desenvolver uma aplicação Web, todas elas irão oferecer uma forma de se acessar as informações da requisição, podendo ter mais facilidades e funcionalidades prontas, isso é a única coisa que vai variar no código de um ou outro, além da própria linguagem.

Também é preciso pensar em como a aplicação funcionará em produção, para saber o que pode influenciar de acordo com a configuração do servidor HTTP.

Exercícios

1. Crie um script CGI com uma página HTML, teste o acesso com `Content-Type: text/html` e `Content-Type: text/plain`, vendo a diferença na renderização do navegador.
2. Crie um script WSGI com uma página HTML.

Discussão

1. Como fazer para gerar páginas diferentes a partir do mesmo sccript WSGI?
2. Como enviar imagens ou outros arquivos estáticos através do WSGI? Isso é eficiente?

Capítulo 03 - Código executado no navegador

Uma aplicação Web é composta de vários elementos, como textos, imagens, efeitos visuais e código. Todos esses elementos encontram-se no servidor, bastando o cliente requisitá-los para que a aplicação possa ser executada no

navegador.

HTML

O HTML é uma linguagem de marcação utilizada para descrever o conteúdo das páginas, além de indicar o endereço dos recursos complementares. Para que tal tarefa seja executada, esta linguagem é composta de marcações, também conhecidas como *tags*, que indicam o início e fim de um menu, texto, parágrafo, link e qualquer outro elemento que se possa imaginar.

O estudo das *tags* HTML, bem como seus atributos, está além do escopo deste texto. Porém existem excelentes documentações na internet sobre este assunto, como a MDN (Mozilla Developer Network), que apesar de ter o nome da Mozilla que desenvolve o Firefox, descreve o padrão e peculariedades de outros motores Web além do Gecko (utilizado no Firefox), como o Webkit, que é a base de muitos navegadores.

Uma relação das *tags* do HTML pode ser encontrada em https://developer.mozilla.org/en-US/docs/Web/HTML/Element, e uma relação dos atributos pode ser encontrada em https://developer.mozilla.org/en-US/docs/Web/HTML/Attributes.

Também é importante observar que o HTML tem duas sessões distintas, `head` e `body`. No `head` encontra-se algumas informações como título da página, ou meta-informações, porém que não aparecem diretamente na página. Diferente do `body` que contem o conteúdo da página em si. Um exemplo de página HTML na sua versão 5, pode ser vista a baixo:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Teste</title>
    <meta charset="utf-8">
  </head>
  <body>
    <h1>Teste</h1>
  </body>
</html>
```

CSS

É possível fazer uma página apenas com HTML, porém como ele foi desenvolvido inicialmente para publicação de textos científicos, o visual padrão não é o melhor para todos os tipos de aplicações. Para permitir a customização da forma como os elementos são exibidos na tela foi criado o CSS, que funciona como um guia de estilos, onde algumas propriedades dos elementos são alteradas, quando os mesmos se encaixam nas regras do arquivo CSS, como pode-se ver no arquivo abaixo:

```
body {
  font-family: sans-serif;
  font-size: 12px;
  color: #333333;
  background-color: #ffffff;
}

h1 {
  font-family: serif;
  color: #000000;
}
```

As regras e propriedades do CSS também estão além do escopo deste texto, porém existe bastante material na MDN sobre o assunto.

JavaScript

Com HTML e CSS é possível criar o conteúdo de uma página e alterar sua forma de exibição, porém todas as informações estão estáticas neste arquivos, não permitindo que quem controla o navegador interagir com elas. O JavaScript, diferente das outras duas é uma linguagem de programação, permitindo que coisas mais interativas ocorram durante a exibição de uma página, uma vez que pode ser executado diretamente no navegador do cliente.

Como o código JavaScript é executado durante a exibição de uma página, não é necessário a comunicação com o servidor para fazer alterações na própria página. Um exemplo pode ser visto a baixo:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Soma</title>
    <meta charset="utf-8">
  </head>
  <body>
    <input id="n1" type="number" value="0"> +
    <input id="n2" type="number" value="0"> =
    <output id="res">0</output>

    <script>
      function somar() {
        var n1 = document.getElementById('n1');
        var n2 = document.getElementById('n2');
        var res = document.getElementById('res');

        res.value = parseInt(n1.value) + parseInt(n2.value);
      }

      (function() {
        document.getElementById('n1').addEventListener('change', somar);
        document.getElementById('n2').addEventListener('change', somar);
      }());
    </script>
  </body>
</html>
```

Como pode ser visto, o JavaScript faz alterações na página, buscando os elementos do HTML, esse processo chama-se manipulação de DOM, uma vez que a cópia da página HTML que está em execução no navegador recebe o nome de DOM.

Atualmente a única linguagem de programação que é possível sua execução nos navegadores é o JavaScript, porém existem alguns projetos como o CoffeScript, Brython e outros que ou são bibliotecas em JavaScript ou compilam para JavaScript, permitindo assim sua execução no navegador. Porém como no final o que está sendo executado é JavaScript, tendem a serem mais lentas e criam uma camada a mais que pode dificultar a identificação de problemas.

Com essas linguagens que compilam para JavaScript, nem sempre o código resultante é muito legível por humanos, dificultando o entendimento do que realmente está sendo executado no navegador, podendo assim também ocultar código malicioso.

Ajax

O fluxo inicial de carregamento de uma página é o cliente requisitar o HTML, olhar a resposta e quando identificar o endereço de recurso externo a mesma (imagens, CSS e JavaScript por exemplo), requisitá-lo. Isso gera uma grande comunicação quando a página está carregando, porém uma vez que ela foi carregada não ocorrem mais requisições.

Com Ajax isso é diferente, via código JavaScript é possível criar novas requisições, como quando um botão for pressionado, ou formulário for preenchido, porém sem ter que recarregar toda a página. A principal vantagem é que essas requisições ocorrem em *background* e o usuário pode continuar interagindo com a página enquanto isso.

O código abaixo mostra o carregamento dinâmico da página utilizando Ajax pela biblioteca jQuery:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Ajax</title>
    <meta charset="utf-8">
  </head>
  <body>
    <ul>
      <li id="pg1">Página 1</li>
      <li id="pg2">Página 2</li>
    </ul>
    <div id="conteudo"></div>
    <script src="jquery.min.js"></script>
    <script>
      function loadPage(page) {
        return function(evt) {
          $.ajax({
            method: 'GET',
            url: page,
            success: function(date) {
              document.getElementById('conteudo').innerHTML = date;
            }
          });
        }
      }
      document.getElementById('pg1').addEventListener('click', loadPage('pg1.html'));
      document.getElementById('pg2').addEventListener('click', loadPage('pg2.html'));
    </script>
  </body>
</html>
```

Acompanhar a execução no navegador

Um recurso que foi desenvolvido e que foi desenvolvido e integrado a pouco tempo é a possibilidade de fazer o acompanhamento da execução da aplicação Web diretamente no navegador. Firefox, Chrome e Internet Explorer possuem suas ferramentas para desenvolvimento, podem variar as funcionalidades e recursos disponíveis, porém envolve visualizar e manipular o DOM, CSS e JavaScript, além de acompanhar as requisições HTTP efetuadas, bem como suas respostas e tempo para identificar possíveis problemas de desempenho.

A utilização destes recursos no Firefox é extremamente simples, bastando pressionar `Ctrl + Shift + I` para abrir as ferramentas de desenvolvimento. Também é possível clicar com o botão direito em cima de um elemento da página e na opção `Inspecionar elemento` já abre o DOM com o elemento selecionado. Se alguma funcionalidade da página não estiver funcionando corretamente, a aba `Console` pode trazer alguma mensagem de erro que ajude a identificá-lo e corrigi-lo.

Exercícios

1. Crie uma página HTML com arquivos CSS e JavaScript separados do HTML principal e acompanhe as requisições na aba `Rede` do Firefox.
2. Faça alterações na página HTML através do JavaScript e da aba `Inspecionar` do Firefox.
3. Faça requisições Ajax e acompanhe as mesmas através da aba `Rede` do Firefox.

Discussão

1. Qual a vantagem de ter os arquivos CSS e JavaScript juntos no HTML? Qual a vantagem de deixa-los separados?
2. O que são imagens *inline*?
3. O que é a minificação do JavaScript? Quais suas vantagens e desvantagens? Esse processo é reversível?
4. Por quê páginas que usam Ajax funcionam apenas no servidor, enquanto páginas sem Ajax funcionam abrindo o HTML diretamente no navegador?
5. Qual o motivo da frase abaixo ser falsa?

   Links com `/imagens/foto.jpg` só funcionam quando abertos via um servidor, enquanto links como `imagens/foto.jpg` funcionam tanto com servidor, quanto abrindo o HTML diretamente.

Capítulo 04 - Utilizando APIs

Com o Ajax, além de carregar partes dinamicamente na aplicação Web, possibilitou a utilização do que é chamado de API, como determinados endereços para efetuar cadastros, consultas, edição e exclusão. De forma geral, quando um serviço é exposto por uma API, vários programas podem interagir com ela, a fim de oferecer uma interface para o usuário executar suas atividades.

API REST

Uma API REST é uma API que segue alguns princípios, como endereços simples e intuitivos para acesso a determinado recursos, utilização do método da requisição HTTP para determinar a ação a ser efetuada, e assim por diante.

Alguns exemplos de endereços podem ser vistos a baixo:

Método	Endereço	Descrição
GET	/clientes	Listar clientes cadastrado no sistema.
POST	/cliente	Cadastra um novo cliente.
GET	/cliente/1	Busca os dados do cliente número `1`.
PUT	/cliente/1	Altera as informações do cliente número `1`.
DELETE	/cliente/1	Apaga o cliente número `1`.

Nestes endereços os dados do cliente, por exemplo no cadastro, podem ser enviados para o servidor através de parâmetros da URL ou no corpo da requisição.

O corpo da resposta destas requisições podem ser uma página com as informações, porém o mais comum é a utilização de alguma linguagem facilmente lida por computador, uma vez que permite o acesso de uma informação diretamente. Linguagens de marcação são ótimas para isso, como o XML, porém esta perdendo espaço para o Json, que é a notação de objetos do JavaScript. Desenvolvendo uma aplicação Web, que terá código JavaScript, se a API responder em Json, muitas coisas serão facilitadas, como a transformação da resposta em Objetos do JavaScript.

Templates JavaScript

Uma API REST na maioria dos casos retorna apenas as informações, ou seja, quem fez a requisição deve tratá-los para fazer sua exibição. A principal vantagem para o servidor é que ele não terá que fazer este processamento. A transformação dos dados em uma página HTML, embora rápida, quando feita pra milhares de páginas representa uma carga considerável no servidor, transferir este trabalho para o cliente pode ser uma boa estratégia.

Partindo deste princípio, surgiram diversas linguagens de templates para JavaScript, sujo a principal funcionalidade é receber dados e transformá-los numa página HTML. Isso quando feito dentro do navegador permite uma alternativa interessante junto com a API REST em relação a geração da página no servidor.

Capítulo 05 - Arquitetura de aplicações Web

Além do código em si e como será executado no cliente, existem várias extratégias de como será disponibilizado pelo servidor.

Servidor

Além da discussão de qual programa de servidor HTTP será utilizado, também é possível criar arquiteturas com diversos serviços. Alguns exemplos serão discutidos na sequência.

Servidor Único

A estratégia mais simples é a configuração de um único servidor, que responderá a todas as requisições. Por não ter muitos elementos, pode ser a forma mais fácil e rápida de disponibilizar uma aplicação. Porém pode apresentar problemas quando a quantidade de requisições aumentar muito, e ter dificuldades para escalar depois, quando se pensar em configurar um segundo servidor para fazer balanceamento de carga.

Proxy reverso e balenceamento de carga

Outra estratégia é utilizar um ou mais servidores para receber as requisições, porém em vez de processá-las, eles enviam-nas para diferentes servidores que fazem o trabalho, possibilitando o balanceamento das requisições. Também é possível não utilizar esses servidores que distribuiem as requisições e fazer isso via DNS, ou utilizar o DNS para fazer o balenceamento para esses servidores. Porém a vantagem de se ter servidores para distribuir as requisições é a posibilidade de já responder arquivos estáticos, que não precisam de processamento, ou ainda manter algumas respostas em cache, reduzindo a quantidade de requisições que realmente precisam serem processadas.

Os servidores que geram as páginas dinâmicas podem responder a diferentes partes da aplicação, sendo um caso mais simples, porém ainda poderia apresentar os problemas de escalabilidade da estratégia de servidor único. Também é possível ter diferentes servidores que podem gerar as mesma partes, e assim ter maior escalabilidade.

Como nem sempre é possível ter certeza que todas as requisições de um cliente irão para o mesmo servidor, a utilização de recursos locais do servidor ou a guarda de estado local não é recomendável, já que uma vez que a requisição chegue em outro servidor, estes recursos não estarão disponíveis. Desta forma a ideia de serviços é uma ótima opção, já que todos esses servidores podem utilizar o mesmo serviço e assim compartilhar as informações.

Esta ideia de múltiplos servidores com proxy reverso e utilização de serviços é fortemente baseada em PaaS, como os baseados em CF, ou até mesmo contâiners. Para melhor entendimento de como uma aplicação adaptada para PaaS deve ser desenvolvida existe o texto 12 fatores.