

# Software Development with Scripting Languages: Web Applications

Onur Tolga Şehitoğlu

Computer Engineering,METU

30 April 2012

- 1 Introduction
- 2 Web application
  - Advantages
  - Major Issues
- 3 HTTP Requests
  - HTTP Methods
  - HTML form
- 4 CGI
  - Application Server
  - Mod Python
- 5 WSGI
- 6 Session and Application Lifetime
- 7 Cookies

# Introduction

- Idea initiated in early 90's for collaboration over Internet.
- World-wide-web: in general meaning applications with document access around Internet
- specifically: technologies around **HTTP** and **HTML**
- HyperText Transmission Protocol: A protocol to transmit documents and provide browser based access to Internet applications.
- HyperText Markup Language: rich text description format with hyperlinks to other documents and data.
- Early application: static access to hyperlinked content
- Later: server based applications accessed by the browser
- Web 2.0: dynamic content, user collaboration, interoperability, b2b links
- Web 3.0: computer generated content, AI, semantics.

# Web application

- client is a web browser and communicates through HTTP to server.
- Early examples: user interface is HTML, CGI (Common Gateway Interface) is used to transmit data.
- Later: XML based data description, stylesheets in visualization (CSS, XSLT).
- DHTML, XHTML, Javascript.
- Trend: Modelling server side as services. Browser side: heavy use of browser side scripting. (ExtJS, jQuery, Google Web Toolkit,...)

# Advantages

- Portability: anyone with a decent browser and Internet can use the application.
- Ease of deployment: no software installed on client side.
- Customizable: personalization for each user is possible.
- Interoperable: server to server communication, distributed architectures are possible.
- Scalability and utilization: Load balancing on the server side is possible.

# Major Issues

- HTTP is connectionless: authentication, continuity, persistence, transaction management is a problem
- Central application model, efficiency is essential.
- Browser capabilities are different.
- Security issues: applications are usually exposed to whole globe.
- Efficiency of HTTP, HTML, Javascript.

# HTTP Requests

- An HTTP v1 request has the following pattern:

*METHOD PATH/URL* HTTP/1.1\r\n

*head1: value*\r\n

...

*headn: value*\r\n

\r\n

*(optional) body*

\r\n

\item HTTP response has the following pattern:

- An HTTP response has the following pattern

HTTP/1.1 *STATCODE STATSTR*\r\n

*head1: value*\r\n

...

*Content-Length: sizeof body*\r\n

\r\n

*response body*

- 200 is success, 3xx redirection, 4xx client errors, 5xx server errors

# HTTP methods

- **GET**: Get content of a web page. Arguments are passed in request URL
- **POST**: Post data to a web page. Arguments are passed in request body
- **HEAD**: Get only headers of a web page.
- **OPTIONS**: Return supported methods
- **PUT**: Upload a representation of a resource
- **DELETE**: Delete a resource
- **CONNECT**: For proxy requests over SSL.
- GET and POST are the most common methods. Others used in special applications.



# HTML form

```
<form method="get or post"
      action="cgi link here"
      enctype="application/x-www-form-urlencoded or multipart/form-data">
<input type="text" name="name"/>
  Sex: male <input type="radio" name="sex" value="male"/>
        female <input type="radio" name="sex" value="female"/> </br>
        student? <input type="checkbox" name="student"/>
</form>
```

- **GET**: form values encoded in the URL separated by & :  
.../post.cgi?name=onur+tolga&sex=male&student=checked
- **POST/x-www-form-urlencoded**: form values are given in a single line in the body of the request packet.

```
POST .../post.cgi HTTP/1.1
```

```
Host: ceng.metu.edu.tr
```

```
Content-type: application/x-www-form-urlencoded
```

```
Content-length: 40
```

```
name=onur+tolga&sex=male&student=checked
```

- **POST/multipart/formdata**: form values are given in MIME multipart form (usefull in large content like file upload)

```
POST .../post.cgi HTTP/1.1
```

```
Host: ceng.metu.edu.tr
```

```
Content-type: multipart/formdata ; boundary=-----abc--
```

```
-----abc--
```

```
Content-Disposition: form-data; name="name"
```

```
onur tolga
```

```
-----abc--
```

```
Content-Disposition: form-data; name="sex"
```

# GET or POST?

- GET is cached, POST not
- When Back button pressed, POST needs resubmitting the data
- GET displays parameters in URL, looks crowded
- GET can bookmark a URL with its parameters
- GET has a URL size limit for parameters (2048 bytes)
- GET only supports ASCII, web encoded parameters, POST can send binary data
- GET is insecure, i.e. passwords will be visible

# Common Gateway Interface

- How server side program interacts with browser and web server
- Traditional CGI: server starts a new external process per request.
- Fast CGI: a single process is created to handle multiple requests.
- Still scalability issues.
- Embed interpreters into server process. Only scripts are loaded, not whole processes. `mod_perl`, `mod_php`, `mod_python`
- Each server worker (process or thread) has an interpreter and interpreter loads the script.

## CGI data flow

- 1 Browser sends data to server in HTTP request (GET or POST)
- 2 Server starts the application instance (external program or script instance) passing the form data
- 3 Form data is passed in environment variables (GET) or as standard input to the application.
- 4 Application reads, parses the input and constructs the HTTP response

# Application Server

- Web server written in native language (Java, Python, Javascript)
- Each connection is handled by a new instance of an application class.
- Advantages like persistence, session management.
- Web server is not as efficient as one developed with C/C++.
- Besides Java not much standart.

# Mod Python

- Apache module allowing python scripts to be executed directly
- A single interpreter loads/processes python scripts
- Server and post data is made available through request objects and parameters.
- Not actively maintained, frozen for 3 years for inactivity.

# WSGI

- Python Web Server Gateway Interface
- Standardization of web servers supporting Python and applications in Python
- Any server/middleware works with WSGI applications and WSGI compliant frameworks
- Many server libraries, middlewares and server supports including **Apache mod-wsgi**
- Many applications/frameworks run with WSGI.  
WSGI application can run on various servers.  
WSGI capable server can run all WSGI applications.



# WSGI Application

- WSGI needs a callable (a function or any object implementing `__call__`)
- Passes environment a callback function to callable
- Callable calls callback with HTTP status string with response to send headers
- Then returns the body of the content

```
def application(environ, start_response):  
    response_body = "<html><body>Hello World</body></html>"  
    status = '200 OK'  
    response_headers = [('Content-Type', 'text/html')]  
    start_response(status, response_headers)  
    return [response_body]
```

# WSGI server

- Reference WSGI implementation contains a simple web server

```
from wsgiref.simple_server import make_server

httpd = make_server('0.0.0.0', 8000, application)
print "Serving on port 8000..."
httpd.serve_forever()
```

- `application` is the callable function of your application

# Getting Input

- Input of web application is GET and POST form data from browser
- WSGI pass it in environment object (first parameter)
- If method is GET it is in `env['QUERY_STRING']`
- If method is POST it is read from `env['wsgi.input']` ( a file object)
- Form data can be parsed by `cgi.parse_qs` from `cgi` module

- An HTTP request/response has one shot lifetime. Each connection is independent and closed after a short period.
- Each time a user clicks a button or link a new application instance has to get that request.
- Keeping a user session open requires extra mechanisms
- Browsers support persistent headers that are called **cookies**.
- A **Cookie** is a variable that can be set by the application in HTTP response
- After a cookie is set, the browser keeps sending same variable value to the same application. (not accross servers!!)
- An application typically associates a session variable with users state and all instances load state from variable

# Cookies

- A Cookie is set through **Set-Cookie** header by the server response. Browser sends cookie to **same domain only!**
- Each cookie header sets a **varname=value** information and optional attributes:
  - **Domain** like `metu.edu.tr` for cookies shared by different domains. This cookie set by `www.metu.edu.tr` will be sent to `webmail.metu.edu.tr` as well.
  - **Path** like `/app` for same server has multiple applications. Only URLs prefixed by the Path will take the cookie.
  - **Expires** the time for expiration of a cookie in a standard form: `strftime("%a, %d %b %Y %H:%M:%S %Z", gmtime(time() + age))` can be used to set a cookie with given **age** in seconds.
  - **Secure**: cookie will only used by browser in encrypted connections.
  - **HttpOnly**: cookie will be used only for direct HTTP/HTTPS requests. Scripts cannot send this cookie.

- A sample header would have been:

```
Set-Cookie: themepref=highlight; Domain=metu.edu.tr; Path=/;  
Expires=Fri, 15 Dec 2017 07:08:03 GMT; HttpOnly
```

- The browser sends this cookie in requests headers to all servers with \*.metu.edu.tr and metu.edu.tr until expiration date. Only user initiated requests sends this cookie (HttpOnly). Cookie header will be:

```
Cookie: themepref=highlight
```

- If more than one cookie is to be sent to the same domain, they are sent in a semicolon separated list:

```
Cookie: themepref=highlight; sessid=aa5a2ba151sdaqg2; page=32
```

## Third-party Cookies

- Domain A can send an irrelevant cookie for another domain B with `Domain` attribute.
- A user accessing domain `xshopx.com` can set a cookie to `example.com`.
- When user visits `nvsprr.com`, it makes a request to `example.com` in a frame or in a browser script.
- `example.com` gets the same cookie and shows a custom advertisement for user using his history in `xshopx.com`.
- There are legitimate uses like third party authentication, but mostly for advertisement and user tracking.
- Some browsers disable them by default.

# Setting session

- 1 If no session variable is set ([Cookie](#) header):
  - 1 Show login page and authenticate
  - 2 On authentication response, set session cookie ([Set-Cookie](#), give a random id and save on database)
  - 3 redirect to original page (302 [Found](#) result, [Location](#) header in HTTP response)
- 2 If session variable is set, load the session variable and state from database