Sofware Development with Scripting Languages: Web Applications

Onur Tolga Şehitoğlu

Computer Engineering,METU

30 April 2012

Introduction

- Idea initiated in early 90's for collaboration over Internet.
- World-wide-web: in general meaning applications with document access arround Internet
- specifically: technologies arround 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, XLST).
- 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 descent 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="check" 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 stdandart 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
- Standartization of web servers supporting Python and applications in Python
- Any server/middleware works with WSGI applications and WSGI complient 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 standart 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 seperated list:

  ```
  Cookie: themepref=highlight; sessid=aa5a2ba151sdaqq2; 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 nvsppr.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 valid session variable is set (Cookie header):
   1. Show/redirect login page and authentication form.
   2. On authentication form post, set session cookie (Set-Cookie header, 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 state from database

Model View Controller

- Model: current state of the application. Usually kept on database for Web applications.
- View: the HTML page displayed by the browser.
- Controller: Handlers of form element actions.
- Interactions:
  - Page load → View generates HTML from model.
  - User action → Controller (action handler) updates the model
  - Model updated → View generates/updates the page
  - Without Javascript, all runs on the server side and requires page reload for updated view.
- Separating look and feel from application logic, View and Controller. Template Engines help.
- SQL vs class Interface for accessing the model. Object Relational Mapping.

RESTful API

- Representational State Transfer
- Not specific to HTTP, but is naturally applies to HTTP.
- Principles:
  - Uniform interface
  - Client-server
  - Stateless
  - Cacheable
  - Layered
  - Code on demand (optional)
- RESTful is an architecture designed for simplicity, scalability and inter-operability

Principles

- Uniform interface
  - Resource addresses with URIs (ie. HTTP URIs)
  - Similar methods for manipulation and access (HTTP methods, GET, PUT, POST,...)
  - Self descripted content (Clean URIs and XML, JSON like response).
  - Hyperlinked data. References returned as hyperlinks.
- Stateless
  - Each request contains all necessary information to execute an action
  - Response does not rely on history of action or a state from the component
  - Server does not have to store state.
  - No need to recover states on failure.

- Client-Server
  - Seperation of concerns
  - Model and business logic on the server
  - Presentation in the client
- Cacheable
  - Cacheability is marked in the response with expiration
  - Client can use cached copy during the specified period.
  - Intermediate layers and client can support cache
- Layered System
  - There might be multiple layers in the system
  - A component cannot know if a response is direct or goes through layers
- Code on Demand
  - Optional. Response can include scripts, code and links.
  - ie. An HTML/Javascript with data, video player link with stream URI.
  - Security issues

HTTP RESTful

- Use clean URIs. ie `http://example.com/category/id` stands for `id` in `category`
- HTTP methods can be mapped on uniform operations: GET for retrieve, POST for create, PUT for update, DELETE for delete
- Return a structured and readable format like XML or JSON. Return meaningful field names.
- Return URIs for hyperlinking other resources
- Example: `GET http://example.com/student/123415`

```
{ "sid": "123415",
    "name": "John",
    "surname": "Doe",
    "department": "http://example.com/departments/571/",
    "registered": [
        "http://example.com/courses/ceng315/",
        "http://example.com/courses/ceng350/" ],
}
```

- use JSON or XML representation in POST and PUT methods
- Return cacheability and expiration Cache-control or Expires headers in HTTP.
- Error handling: Use appropriate HTTP result codes (200 success, 403 forbidden, 404 not found, 5xx server erros)
- Provide details in a uniform object as:

```
404 Not Found
Content-Type: application/json

{ "instance" : "/students/123123",
  "detail" : "student id does not exist"
}
```

- POST /student/12314
  Content-Type: application/json

```
{  "name": "Onur Tolga", "surname": "Sehitoglu",
   "department": 599, "email": "onur@ex@mple.com@tr"
}
------
403 Invalid Form
Content-Type: application/json

{ "instance" : "/students/12314",
  "detail" : "student information is not valid",
  "errors": [ {"field": "department",
                "issue": "no such department"},
              {"field": "email",
               "issue": "invalid email format"}],
}
```

REST Authentication

- Stateless and Session contradicts by definition.
- Cookies and plain HTTP authentication can be used in human/browser clients
- Scripts and machine to machine interface is different
- A typical scenario can be:
  - Ask for an authentication token, provide credentials, passwords, signed challenge etc.
  - Server sends a token, valid in a period
  - All following requests include the token either as a cookie or HTTP header.
- The security of authentication token is an issue. (hijacking)
- For a full stateless operation:
  - server stores client public key
  - Client signs all requests (body and URI) and includes in a header
  - Server validates signatures before each requests