Databases

Web programming

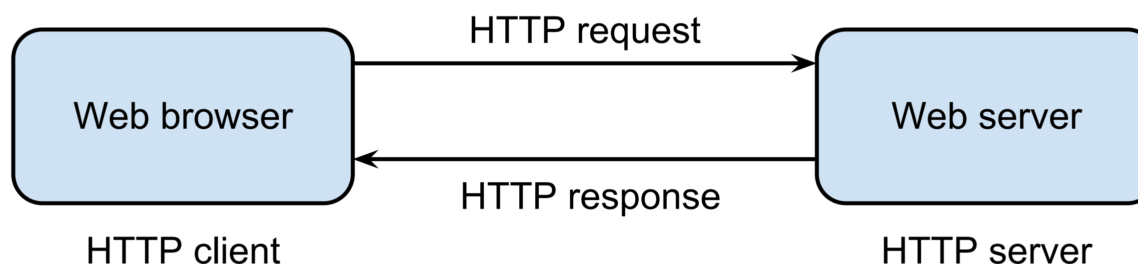
Hallgrímur H. Gunnarsson

Basic model

Whenever your browser fetches a file (web page, image, etc) from a web server, it does so using HTTP (Hypertext Transfer Protocol)

Example: web browser sends a request: give me /abc.jpg or /index.html, and the web server sends back a response: here's the file, followed by the file itself

Formally: HTTP is a request-response protocol based on the client-server model



HTTP request message

HTTP request format:

```
Request line, such as GET /logo.png HTTP/1.1
Zero or more header lines
An empty line
An optional message body (e.g. query data or query output)
```

Request line:

```
METHOD URI HTTP/x.x
```

e.g.

```
GET /path/to/file HTTP/1.1
```

Request methods: GET, POST, HEAD, ... and more

Request method: GET

GET is for getting (retrieving) data (not everyone follows this convention though)

Strictly speaking, GET should be idempotent, i.e. have no side-effects. Execution of $N > 0$ identical GET requests should be the same as for a single request

Therefore, use GET when the request does not change anything, otherwise use POST

Example of a HTTP GET request:

```
GET / HTTP/1.1
Host: www.mbl.is
<empty line>
```

Request method: POST

A GET request is just for getting, but a POST request may involve anything, like storing or updating data, i.e. POST usually involves side-effects

In a GET request, data must be encoded in the URL, e.g.

```
GET /test.php?key1=value1&key2=value2
```

but in POST the data appears within in the message body

Example of a HTTP POST request:

```
POST /test.php HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Content-Length: 23

key1=value1&key2=value2
```

Typical HTTP GET response

Typical response is 200 OK:

```
hhg@hhg:~$ nc 130.208.143.96 80
GET /~hhg4/x.php HTTP/1.1
Host: brainfuck.nord.is

HTTP/1.1 200 OK
Date: Thu, 22 Sep 2011 20:53:02 GMT
Server: Apache/2.2.9 (Debian) PHP/5.2.6-1+lenny13
X-Powered-By: PHP/5.2.6-1+lenny13
Content-Length: 7
Content-Type: text/html

123456
```

HTTP redirect

301 redirect to another location (specified by Location in header)

```
HTTP/1.1 301 Moved Permanently
Date: Thu, 22 Sep 2011 18:57:27 GMT
Server: Apache
Location: http://www.mbl.is/frettir/
Cache-Control: max-age=300
Expires: Thu, 22 Sep 2011 19:02:27 GMT
Vary: Accept-Encoding
Content-Length: 294
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>301 Moved Permanently</title>
</head><body>
<h1>Moved Permanently</h1>
<p>The document has moved <a href="http://www.mbl.is/frettir/">here</a>.</p>
<hr>
<address>Apache Server at www.mbl.is Port 80</address>
</body></html>
```

HTTP is stateless

HTTP is a stateless protocol. Each request is processed independently of other requests

A stateless protocol does not require the server to retain session information or any other state across multiple requests

Contrast with e.g. FTP: each connection is a session with state such as who is logged in, working directory, etc.

Consequently, in order to keep state across multiple requests in HTTP (e.g. to implement *sessions*), the application must store the state, it cannot rely on HTTP

But how can the application track which requests belong to the same session? By using so-called *HTTP cookies*

Keeping state

Main idea: encode state in HTTP server response such that the next request includes the state

Many options: use hidden fields in HTML forms, parameters in URLs, etc. But the most popular one is using so-called *HTTP cookies*

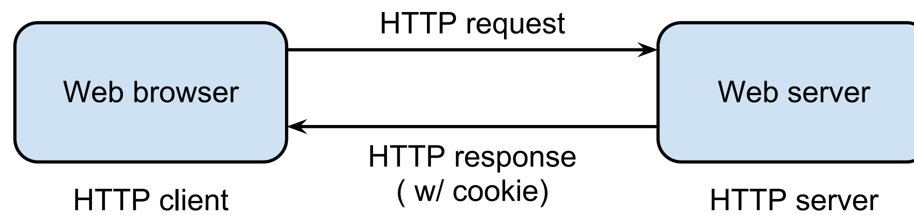
HTTP cookies were invented by Netscape in 1994, and they are implemented in all browsers today

Cookies are key-value strings, issued by HTTP servers as part of a HTTP response

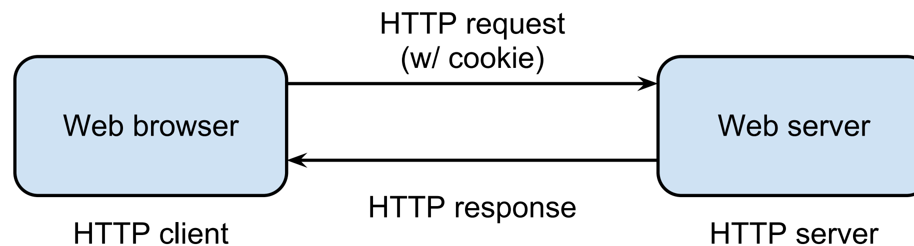
When the HTTP client next communicates with the server, all cookies from that particular server are automatically submitted as part of the HTTP request

Cookie example

Server issues cookie in HTTP response:



Next request includes cookie in HTTP request:



Cookie example

Server issues cookie:

```
hhg@hhg:~$ nc 130.208.143.96 80
GET /~hhg4/x.php HTTP/1.1
Host: brainfuck.nord.is

HTTP/1.1 200 OK
Set-Cookie: PHPSESSID=95bff8aa1946a8ae82933f0b89f03bd5; path=/
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Content-Length: 0
Content-Type: text/html
```

Next request includes cookie:

```
GET /~hhg4/x.php HTTP/1.1
Host: brainfuck.nord.is
Cookie: PHPSESSID=2777a01a242541a3d28f82a399b690f1

HTTP/1.1 200 OK
Content-Length: 0
Content-Type: text/html
```

Sessions

Most common use of cookies is to implement so-called *sessions*

How it works:

1. Web server generates a hard-to-guess session identifier and issues it as a cookie
2. Browser includes the session identifier in all subsequent requests
3. Web server uses the session identifier to store and lookup session state on the server

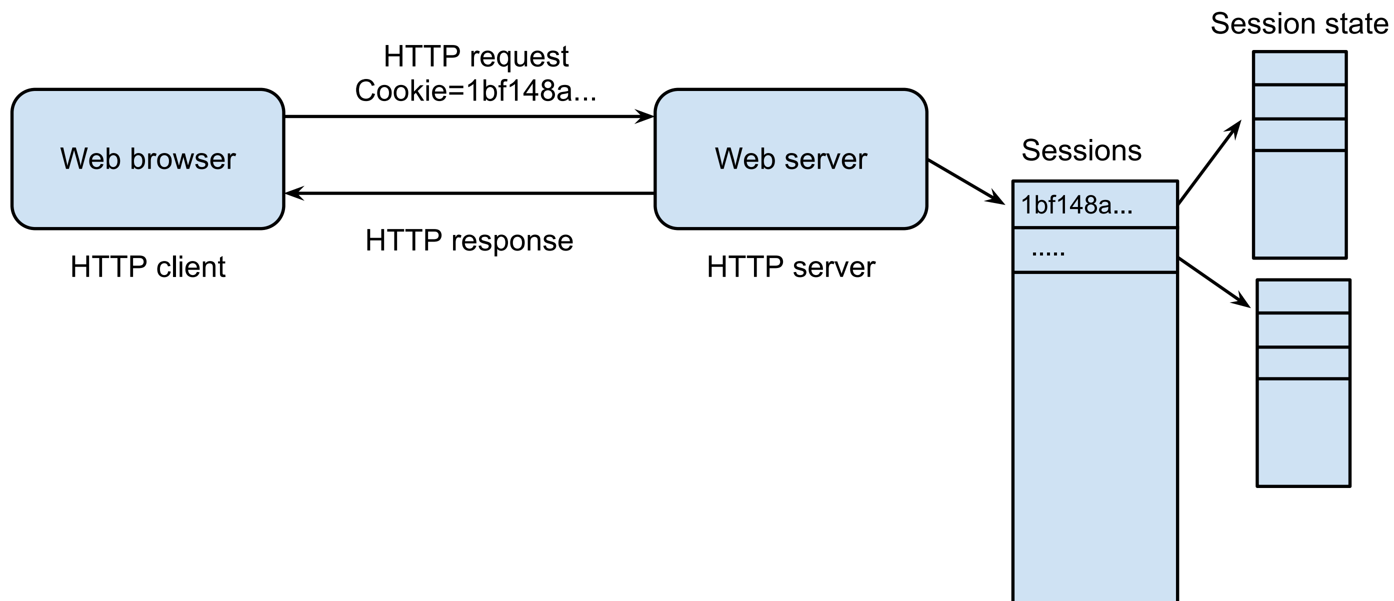
Session state might include whether, for example, a user is logged in, his username, contents of his shopping cart (for a shopping site), etc.

Sessions

Session table maps session identifier $\longrightarrow$ session state

Session state is usually also a table of key/value pairs

Example:

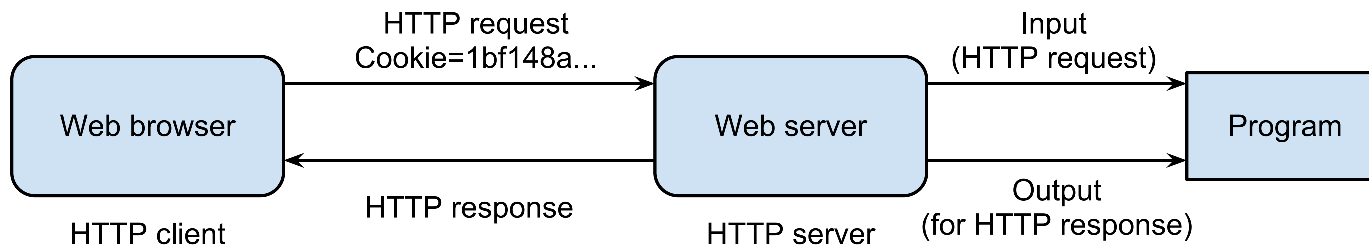


Static vs. dynamic content

Web server can serve *static data*, e.g. files from disk. But it might also serve *dynamic data*, generated on-the-fly in response to a request

Usually dynamic data is generated by executing code in some form on the server

Executed code can access data from HTTP request, e.g. parameters and data



Basic PHP code

Default PHP setup in Apache works like this: if the filename ends in ".php", then the file is executed using the PHP interpreter

Code is embedded in file, enclosed in special <?php and ?> tags.

```
<html>
<body>
<h1>Testing</h1>
<?php
print "Hello world\n";
?>
</body>
</html>
```

Sessions in PHP (1)

Calling the session_start() function will open (and if necessary create) a persistant session on the server

The session is identified by using a cookie named PHPSESSID

```
<html>
<body>
<h1>Testing</h1>
<?php
session_start();
print "Hello world\n";
?>
</body>
</html>
```

Sessions in PHP (2)

Session state can be accessed through the global variable $_SESSION in PHP

$_SESSION is a dictionary (associative array) of key-value pairs

```
<html>
<body>
<h1>Testing</h1>
<?php
session_start();
print $_SESSION['counter'];
print "\n";
$_SESSION['counter'] += 1;
?>
</body>
</html>
```

PHP walkthrough

Code walkthrough (all code will be available on the website):

1. Login/logout example
2. HTML forms
3. Accessing GET/POST data from PHP
4. HTML form -> SQL insert example
5. SQL query from PHP, render SELECT as HTML table
6. Using templates to separate code from HTML
7. Using template to render SQL result

Caching

Basic idea: If fetching some value is expensive (and you might need to fetch it repeatedly), then fetch it once, store it in memory and use it often

Memory vs. time trade-off

Caching is everywhere, CPU caches, page cache (disk cache), web cache, DNS cache, database cache, etc.

In the context of web programming and databases: database is slow, rendering might be slow $\Rightarrow$ cache database results, pages, etc.

Why use caching?

Disk is the new tape! Fast for sequential access, very slow random access

DRAM: 100ns to access, 20 GB/s, $12/GB

Disk: 10ms to access, 200 MB/s, $0.05/GB

SSD: 0.1ms to access, 250 MB/s, $1.5/GB

$\Rightarrow$ Anything that touches disk kills performance

Numbers everyone should know

execute typical instruction	1/1,000,000,000 sec = 1 nanosec
fetch from L1 cache memory	0.5 nanosec
branch misprediction	5 nanosec
fetch from L2 cache memory	7 nanosec
Mutex lock/unlock	25 nanosec
fetch from main memory	100 nanosec
send 2K bytes over 1Gbps network	20,000 nanosec
read 1MB sequentially from memory	250,000 nanosec
fetch from new disk location (seek)	8,000,000 nanosec
read 1MB sequentially from disk	20,000,000 nanosec
send packet US to Europe and back	150 milliseconds = 150,000,000 nanosec

(Source: Peter Norvig)

Memcached

A open-source high-performance, distributed memory object caching system

Memcached is generic in nature, but it was originally intended to speed up dynamic web applications by alleviating database load

It is an in-memory key-value store for small chunks of arbitrary data (strings, objects) from results of database calls, API calls, page rendering, etc.

Basically, a big hash table in memory with expiration rules, and basic get/set/increment/etc. operations

Big users: Facebook, YouTube, Reddit, Twitter, Zynga

Typical Memcache + Database Usage

Typical fetch operation:

1. Try to fetch data from cache
2. If it exists, return data
3. If it does not exist, query database, store result in cache, return data

Typical update operation:

1. Update database
2. Expire or update value in cache

Memcached at Facebook

Total

Operations:

→ Over 400M GET/s

→ Over 28M SET/s

Size:

→ Over 2T items

→ Over 200TB of RAM

Network I/O:

→ Peak rx: 530Mpkts/s, 60 GB/s

→ Peak tx: 500Mpkts/s, 120 GB/s

Typical server

Operations:

→ 80K GET/s

→ 2K SET/s

Size:

→ 200M items

→ 64GB of RAM

Network I/O:

→ rx: 90Kpts/s, 9.7 MB/s

→ tx: 94Kpts/s, 19 MB/s

Source: memcache@facebook at QCon 2010 in Beijing

Topics to discuss

1. Cache stampede
2. How to partition data in a distributed caching system
3. Hashing and consistent hashing