Database-enabled web technology
Session management & Security

Instructor: Çağrı Çöltekin
c.coltekin@rug.nl

Information science/Informatiekunde

Fall 2011/12

Previous weeks

W1: Quick introductions to PHP & git.

W2: An overview of DB design and SQL.

W3: Some background on server-side programming, HTTP. Interacting with users in PHP: HTML forms, and cookies.

W4: DB Programming: stored procedures, programming with Pear DB, transactions, triggers...

Stored Procedures

- Stored procedures are database-side programs that are stored and run in a DBMS.
- Stored procedures add procedural-language support in relational (SQL) databases.
- Stored procedures are database objects, they are created and dropped the same way as the other database objects.
- Stored procedures run with the credentials of the user who creates them. As a result, one can run stored procedures without having access to any of the underlying tables.
- Stored procedures may reduce the network I/O, and may run faster in certain systems/cases.
- There is a relatively recent standard. However, the stored procedure language differ widely among different DBMSes.

SP in MySQL an example

```
1  drop procedure if exists confirm_order;
2  delimiter $$
3  create procedure confirm_order(in cust_id int, out nitems int)
4  begin
5      declare isbn_tmp varchar(13) default null;
6      declare customer, quantity int;
7      declare more_rows bool default true;
8      declare cur cursor for
9          select cID, ISBN, qty from basket where cID = cust_id;
10     declare continue handler for not found set more_rows = false;
11     set nitems = 0;
12     open cur;
13     fetch cur into customer, isbn_tmp, quantity;
14     while more_rows do
15         set nitems = nitems + quantity;
16         insert into orders (cID, ISBN, qty, order_date, status)
17             values (customer, isbn_tmp, quantity, now(), 'N');
18         fetch cur into customer, isbn_tmp, quantity;
19     end while;
20 end $$
21 delimiter ;
```

```
call confirm_order(10, @nbooks);
select @nbooks;
```


PHP Pear DB library

- Pear DB library provides a unified way of connecting to multiple DBMS systems from PHP.
- In comparison to other methods of database access, e.g., PHP `mysql_` functions, Pear DB provides a more portable approach.
- Independent of the DBMS or library in use, you should always validate the user input.
- Pear DB provides three functions: `escapeSimple()`, `escapeSmart()` and `quoteIdentifier()` to sanitize the input before using in an SQL statement.
- Pear DB also provides a `prepare()`/`execute()` interface (as well as the `query()`).

Pear DB: a first example

```
1  <?php
2      require_once('DB.php');
3      require_once('db-config.php');
4      $conn = DB::connect("mysql://$user:$pass@$host/$db");
5
6      $res = $conn->query('select * from book');
7
8      echo "<table border=\"1\">";
9      echo "<tr><th>ISBN</th><th>title</th></tr>";
10     while ($row = $res->fetchRow(DB_FETCHMODE_ASSOC)) {
11         echo "<tr><td>${row['ISBN']}</td>";
12         echo "<td>${row['title']}</td></tr>";
13     }
14     echo "</table>";
15     $conn->disconnect();
16 ?>
```


DB Transactions

- The (SQL) statements in a transaction is treated as atomic: either all or none of them are run.
- The (SQL) statements in a transaction is treated as isolated: DBMS isolates statements in a transaction from possible effects of other tasks running in parallel.

```
$db->autoCommit(false);
$db->query(...);
...
if (some condition) {
 $db->rollback()
} else {
 $db->commit();
}
```


Today...

- Revisiting cookies.
- Session management.
- Some bits of security: ...

HTTP Cookies

- A cookie is a piece of information a HTTP server asks the client to retain for until a specific expiry date/time.
- Cookies are passed in the HTTP header field (as opposed to GET data in URL, or POST data in content).

The server sends a cookie (in HTTP headers) to a client using,

```
Set-Cookie: name=val; expires=datetime; domain=d; path=p
```

There may be additional options, e.g., `Secure` or `HttpOnly`.
The client sends the matching cookie back in every request if,

- the domain and path matches
- the cookie is not expired

```
Cookie: name=val; name=val; name=val; ...
```


Cookies: domain

- Server can the `domain` of the cookie, by default the domain is the full domain name of the server.
- Client sends back a cookie, only if domain matches.
- If the web server runs on `www.let.rug.nl`, then all cookies for `www.let.rug.nl`, `let.rug.nl` and `rug.nl` will be sent by the client to the server.
- A server is allowed to set cookie domain for its higher level domains (except the top-level domains).
- Cookies for top-level domains, e.g., `.nl`, `.edu`, `.net`, are ignored by the browsers.
- Cross-domain cookies are also discarded by the clients. The server with domain name `www.let.rug.nl` cannot set a cookie for `example.com`.

Cookies: path

- Similar to the `domain`, server can also specify a `path` attribute for a cookie.
- A client sends a cookie if the `path` it requests is a sub-path of the cookie's `path` attribute.
- If path is `/myapp/` then client will sent the cookie back only if it requests `/myapp/` or a sub-path, e.g., `/myapp/login.php`.
- If the cookie path attribute is `/`, then it is sent for all paths in the domain.

Cookies: expiry

- The client retains the cookie until the `expiry` date/time specified during its creation.
- If unspecified, or the value is `0`, the cookie is kept until the browser terminates.
- The server cannot delete a cookie, but it can (re)set the cookie with an expiry time in the past. This will cause the client to delete the cookie.
- Note that the clocks of server and client may not be in sync.

Cookies: options

Besides the name, value and the standard options expiry, path and domain, there are a number of additional options.

- If `Secure` options is specified, the client sends the cookie back only if the connection is secure (HTTPS).
- If `HttpOnly` options is specified, the client does not allow client-side programs (e.g., JavaScript) to access the cookie. Otherwise, the server side programs have access to the cookies in the browser.

Working with cookies in PHP

- You can set cookies with function `setcookie()`. For example `setcookie($name, $val, $exp, $path, $domain, $secure, $httponly)` where, except `$name` all arguments are optional.
- You have to set the cookies before sending any content (remember: they are part of the HTTP headers, not the content).
- Received cookies are stored in the global associative array `$_COOKIE`
- Assuming you have a cookie with name `user`, you can access it using `$_COOKIE['user']`.
- Cookies are also present in the combined associative array `$_REQUEST`

PHP and cookies

```
1  <?php
2      if (!isset($_COOKIE['MyCookie'])) {
3          setcookie('MyCookie',
4                    'some value',
5                    time()+3600*24*7);
6      }
7  ?>
8  <html>
9  <!--- ... some html stuff -->
10
11 <?php
12     if (!isset($_COOKIE['MyCookie'])) {
13         echo "You do not have the cookie yet.";
14     } else {
15         echo "MyCookie = ${_COOKIE['MyCookie']}";
16     }
17 ?>
```


Need for session management

A simple interactive/desktop application

1. process starts
2. displays some output
3. receives some input from the user
4. process ends

A simple web application

1. process starts
2. (possibly) receives some input from the user
3. displays some output
4. process ends

Servers-side programming: difficulties

A server-side web application,

- cannot use ordinary (global) variables that spans throughout the application lifetime.
- has to identify and cope with multiple runs of the same application,
- cannot assume that input provided on the next run is provided by the same source that started the application.

A session in web-based programming provides a way to solve these problems.

What is in a session?

A session consist of two components:

1. A unique session ID passed back-and-forth between client and the server. This makes sure that the server side identifies the client and resumes the session where it was left in the previous step. The session ID can be communicated using:
   - Cookies.
   - 'Hidden' form fields, which passed with GET or POST data.
2. A server side storage for session data. This makes sure that parts of the session can share information. The information is typically stored in local files, but can also be stored in a database (or any other means).

Why not pass all the information back-and-forth like the session ID?

PHP sessions: introduction

- The session is initiated using he function `session_start()`.
- If using cookies for sessions, `session_start()` should be used before any output.
- PHP sessions can use cookies (preferable for most purposes) or GET/POST methods.
- The session information is available through the super global array `$_SESSION`: values of the members of `$_SESSION` persists throughout the session.
- The data is stored in files by default, but other 'handlers' are available, and new handlers can be created by the user.
- Name of the default session ID (cookie or the name of the html form field) is `PHPSESSID`, but can be customized.

PHP sessions: an example

```
1  <?php session_start(); ?>
2  <html> <body>
3  <?php
4      if (!isset($_SESSION['page_seq'])) {
5          $_SESSION['page_seq'] = 0;
6      } else {
7          $_SESSION['page_seq'] += 1;
8      }
9      echo "You are on page ${_SESSION['page_seq']}.";
10 ?>
11
12 </body></html>
```


PHP: starting a session

- `session_start()` starts a session if it is not already started. It will typically send a cookie with default name `PHPSESSID`.
- The default session name can be changed using PHP configuration for the site, or using `session_name()`.
- Cookie parameters, `lifetime`, `path`, `domain`, `secure`, and `httponly` can also be set using `session_set_cookie_params()`.
- Some other session configuration parameters can be configured through `ini_set()` function. A few parameters of interest are (see PHP session manual for more):
  - `session.use_cookies`
  - `session.use_only_cookies`
- Session related parameters must be set before calling `session_start()`.

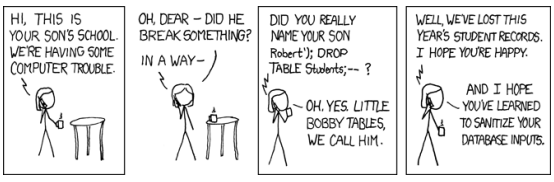
PHP: using session variables

- Session variables are stored in global array `$_SESSION`.
- The members of the `$_SESSION` persists until session is destroyed, by `session_destroy()` or in case the session is expired.
- Session cookie (on the browser side) will by default live until the browser is closed, otherwise it is controlled by `lifetime` of the session cookie.
- `session_name()` (without parameters) returns the name of the session, and `session_id()` returns the session ID.
- Session ID can be changed anytime using `session_regenerate_id()`. It is a good idea to change the session ID at least at every security context change (we will return to this in discussion of security).

Where is my session data?

- PHP keeps your session data, by default, in files in a system-wide directory.
- You can switch to an existing handler, e.g., for sqlite, using PHP configuration variable `session.save_handler`.
- You can write your own handlers, e.g., to keep your session information in MySQL or in memory, using `session_set_save_handler()` You need to specify handlers for: `open`, `close`, `write`, `read`, `destroy`, `garbage collection`.
- This may, for example, allow you to maintain sessions on a load-balanced web server environment.
- Writing your own session handlers may also help you have more control over your sessions.

Web, Databases & Security



http://xkcd.com/327/

Secure coding: why?

An application developed and set up without attention to security, may

- allow unauthorized use of the application,
- provide unauthorized access to a complete system, potentially causing other applications to be compromised,
- leak sensitive information (e.g., passwords, credit card numbers),
- do unintended work for others (typically with malicious intent).

A few guidelines (before we start)

- Always check user input before using (e.g., in an SQL query).
- Do not store and transfer sensitive information unencrypted.
- Do not store or transfer sensitive information if you can avoid it.
- Sanitize your output (e.g., properly escape special characters if you are outputting HTML).
- Try to implement multiple levels/layers of security.

Sessions and Security

Badly implemented session management systems may allow unauthorized access to data/application. Typically,

- An easy to guess session ID may be found by brute-force trial & error.
- An attacker may obtain the session ID by sniffing the network traffic.
- An attacker may steal the session ID/key physically.
- An attacker may trick someone to use a URL (e.g., sent via email), causing a particular session ID to be used (session fixation).

Some guidelines for session security

- Change your session ID frequently, particularly after every authorization level change (e.g., successful login). `session_regenerate_id()` is your friend.
- Avoid using GET, for passing session ID, use cookies when available.
- Use HTTPS, secure cookies if available.
- Timeout your sessions.
- In some cases, you may also consider checking client IP, or referrer string.

Summary & Next week

This week:

- Cookies & sessions.
- Security, particularly related to sessions.

Next week:

- More on security.