Introduction to PHP Part 2
29th October 2010
091 Labs

Author: Barry Coughlan for 091 Labs ::

This week...

- Handling user logins and data: Sessions and cookies
  - Cookies & sessions explained
  - Modifying our database to handle users
  - Creating a login page (code)
  - Checking if the user is logged in (code)
- Web security
  - SQL injection
  - Cross-site scripting (XSS)
  - Cross-site request forgery (CSRF)
- Designing an app from scratch
  - Designing relational databases
    - One-to-many relationships
    - One-to-one relationships
    - Many-to-many relationships
  - SQL JOIN queries

Cookies

Sometimes we want to store data about a particular user. Cookies are files stored on the client machine that contain data.

The client makes a request to log in:

Client (Request):
POST /login.php HTTP/1.1
Host: www.091labs.com
...
user=091labs&password=whatever

The server responds and sets a cookie:

Server (Response):
HTTP/1.1 200 OK
Content-type: text/html
Set-Cookie: username=091labs; token=ab4718ad76928abc; expires=Sun, 1-Jan-2011 00:13:37 GMT;

Cookies (continued)...

Every time a client makes a request (until the expiry time or cookies are cleared) it will send this:

Client (Request):
POST /index.php HTTP/1.1
Host: www.091labs.com
Cookie: username=091labs; token=ab4718ad76928abc;

PHP takes this data and passes it to the .php file through the $_COOKIE['...'] variable.

You can set a cookie in PHP using the built-in setcookie(...) method:

e.g:

```
setcookie("user", "091labs", time()+3600);
setcookie("token", "ab4718ad76928abc", time()+3600);
```

(this cookie expires in an hour (current time in seconds since UNIX epoch + 3600)

To delete a cookie, set it to expire at some time in the past:

```
setcookie("user", "091labs", time()-3600);
```

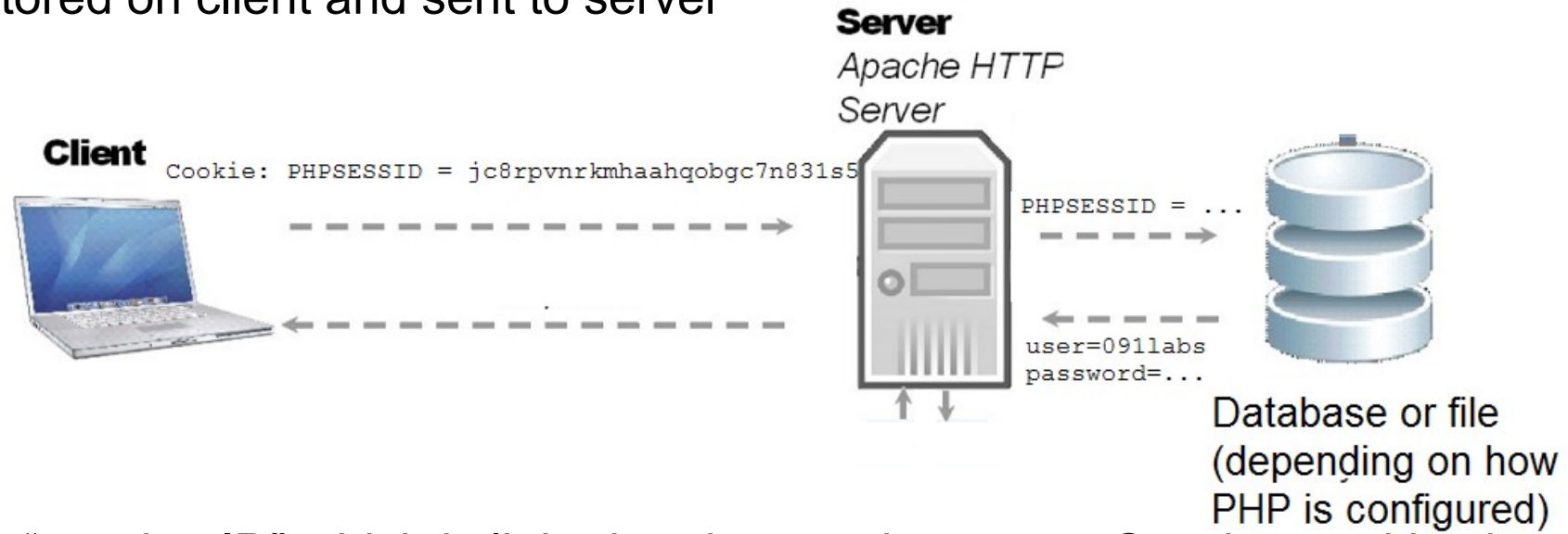
Sessions

The problem: Users can modify cookies and give us false information.
e.g. Online shopping cart, it's not secure to store prices, totals, etc. In cookie

Cookies

User data stored on client and sent to server

Sessions



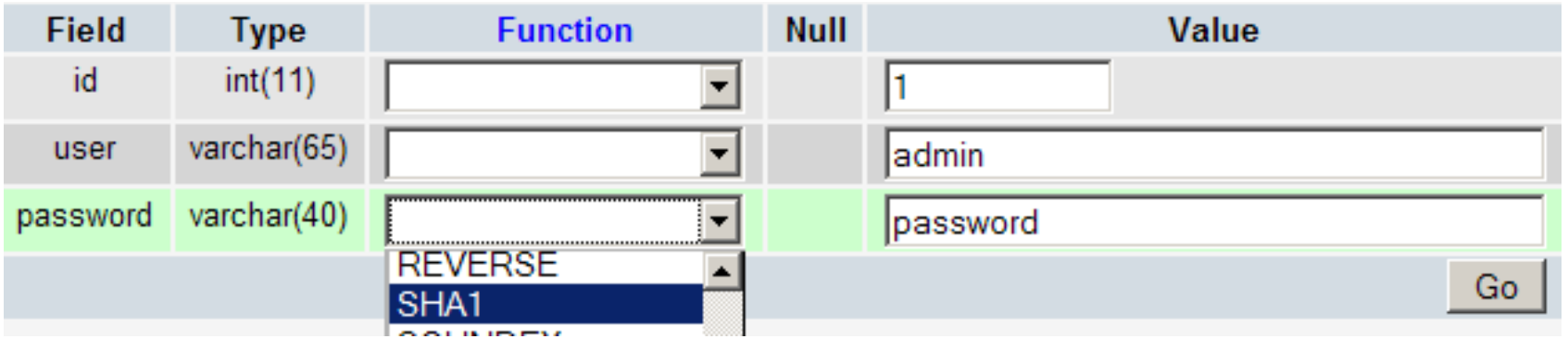
Client has a “session ID” which is linked to data on the server. Session cookies by default have no expiry time (set in php.ini), which in most browsers means the session will end when the browser is closed.

Modifying our database to handle users

Add a table to the 'MyApp' database called “users”:

Field	Type	Collation	Attributes	Null	Default	Extra
id	int(11)			No	*None*	auto_increment
user	varchar(65)	latin1_swedish_ci		No	*None*	
password	varchar(40)	latin1_swedish_ci		No	*None*	

Add a sample user via phpMyAdmin (click the “Insert” link along the top):



Password encryption:

MD5 hashing function used to be very widely used, but vulnerabilities have been discovered in the past few years.

More secure SHA-1, which is 40 bytes in length (our password field must be varchar with length of at least 40 bytes)

Common or simple passwords can still be cracked using Rainbow Tables.

SQL Injection

Suppose we had an SQL query like this:

```
SELECT * FROM users WHERE user='$username' and password=sha1('$encrypted_password')
```

Now suppose that a user entered admin' ;-- in the username box.
(Note: ;-- tells MySQL to ignore everything after that point)

Our query would then look like:

```
SELECT * FROM users WHERE user='admin' ;--' and password=sha1('1234')
```

Which would authenticate the user as “admin” without checking the password.

The solution is to stop users from escaping out of strings. The PHP function **mysql_real_escape_string($string)** escapes all apostrophes by putting a backslash before each one.

admin' ;-- then becomes admin\' ;-- and is then treated as a string by MySQL

Cross site scripting (XSS)

Suppose we had a login page with a URL like:
http://localhost/login.php?username=091labs

With the code:
echo “Thank you for logging in, $username”;

If an attacker put in something like:
http://localhost/login.php?username=<script>alert(document.cookie());
they could inject HTML and javascript code into the page.

This could be used to steal cookies (and impersonate users) among other things.

Prevention:
HTML has special character codes for certain characters. “<” renders as “<” in a web browser. “>” renders as “>”.

The PHP function htmlspecialchars**($string)** replaces all “<” and “>” with “<” and “>”, so that it will not be interpreted as HTML by the browser.

```
$username = htmlspecialchars($_POST['username']);
```

Cross site request forgery (CSRF)

Suppose we had a file for updating our email address, **updateprofile.php:**

1. Check if the user is logged in:

```
<?php
session_start();

if (!isset ( $_SESSION['username'] ) ) {
    header("Location: login.php");
}
```

2. Some function (not shown) for updating the user's email address

```
update_email_address( $_GET["newemail"] );

?>
```

Suppose an attacker embeds this code in a web page:

```
<img src="http://091labs.com/updateprofile.php?newemail=admin@attacker.com">
```

If the user is logged in, their web browser will request the above URL which will change the email address to the attacker's address (which they could then use to reset the password).

Cross site request forgery (CSRF) 2

Protecting against CSRF:

1. Put a random token into the user's session and the form/hyperlink:

```
<?php
    .....
    $_SESSION['token'] = md5(rand()); // MD5 hash of a random number;
?>

<form .....>
    <input name="token" type="hidden" value="<?php echo $_SESSION['token']" />
</form>
```

2. When the form/URL sends, we check the token in the form against the session token.

```
<?php
    if ( empty( $_GET['token'] ) || !strcmp( $_GET['token'], $_SESSION['token'] ) ) {
        die ("CSRF attack detected");
    }
?>
```

Designing relational databases

We use a unique ID (which is called a **Primary Key**) for each row in a table so that we can define relationships between different “entities” (tables) in our database.

Why not just put all the data in one table?

id	forum	thread	user	password	email	post
1	091Labs	Hello!	barry	*****	barry@091labs.com	Hello World
2	091Labs	Hi	barry	*****	barry@091labs.com	How's it going?
3	091Labs	Bye	bob	*****	bob@gmail.com	Goodbye!

If we wanted to update the user “barry”'s email address, we would have to update it in many places. Instead, if we had the user information in another database table, and linked it to the main table we would only have to change it once.

This ensures the **integrity** of the data (so that we don't accidentally end up with two email addresses for “barry”).

Note: Similar things should be done for forum, thread and post here but are not shown.

id	user	pass	email
1	barry	******	barry@091labs.com
2	bob	*****	bob@gmail.com

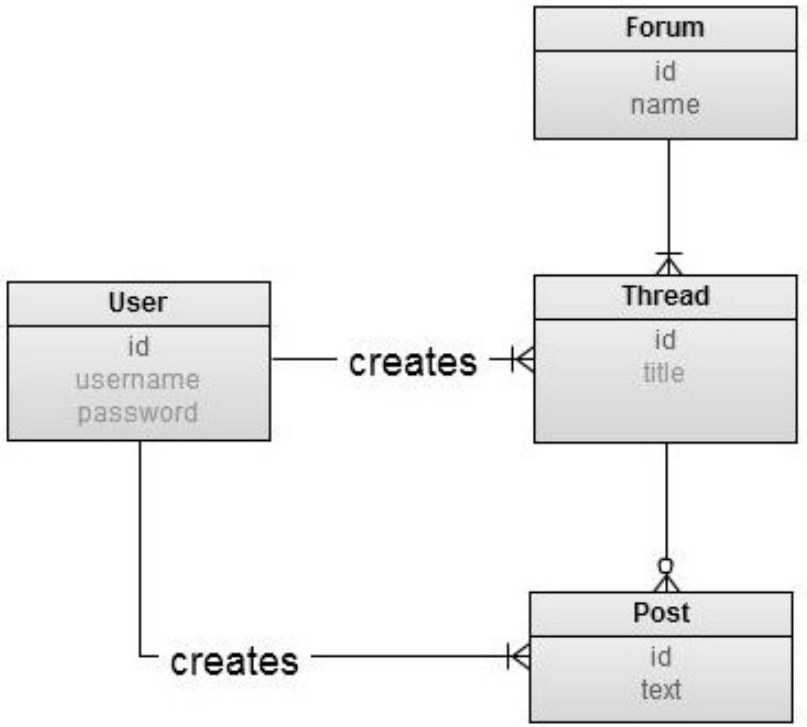
id	forum	thread	user_id	post
1	091Labs	Hello!	1	Hello World
2	091Labs	Hi	1	How's it going?
3	091Labs	Bye	2	Goodbye!

One-to-many relationships

Take a forum as a design example. Each forum **has many** threads, and each thread **has many** posts. Each user is associated with many threads and many posts. Also, each thread or post can be associated with a post (the author of the thread/post).

One-to-many relationships:

A one-to-many relationship says that “X has many Y” or that “X belongs to many Y”. For example, each forum has many threads, each thread has many posts, each user can belong to many threads or posts. Below is an Entity-Relationship diagram, showing these relationships.



How do we link these in our database?

Each thread belongs to one forum/user. Since we only need to store one forum/user value, the best way is to add columns to the “thread” table:

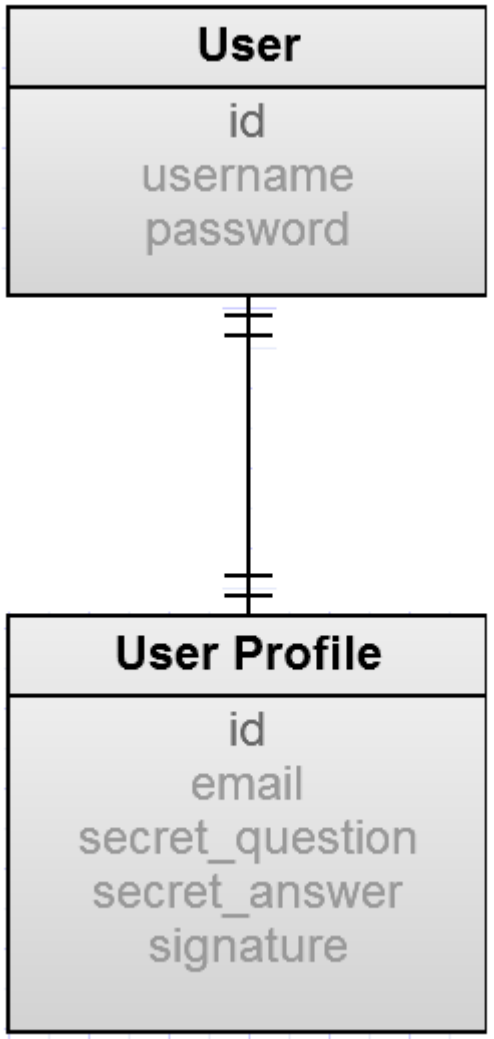
	Field	Type
☐	id	int(11)
☐	user_id	int(11)
☐	forum_id	int(11)
☐	title	varchar(255)

user_id and forum_id are called **Foreign Keys**

As a general rule, in one-to-many relationships the column goes in the “many” table (⊣<)

Note: Some database engines (InnoDB) can be set to enforce referential integrity (e.g. Delete all threads belonging to a forum when a forum is deleted), however this is a bit more advanced and we will do it manually for now to get a feel for relational databases.

One-to-one relationships



One-to-one relationships:

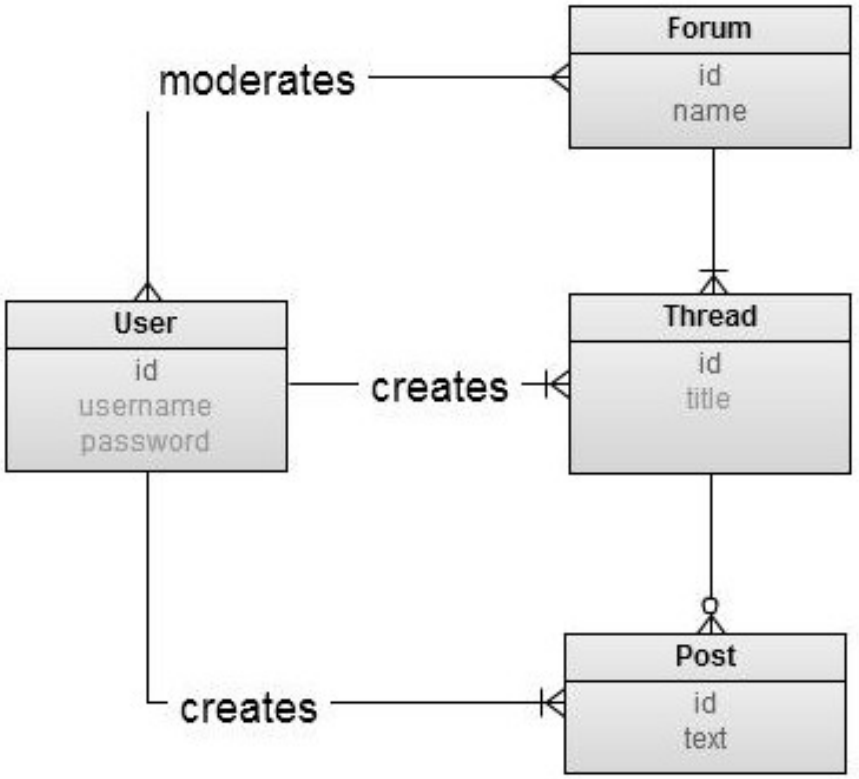
A typical example of a one-to-one relationship is a user profile. A user profile should not be allowed to be associated with more than one user in a database.

One-to-one relationships look the same as one-to-many, except that we set the Foreign Key as unique (in phpMyAdmin)

Putting these restraints in the database helps us catch programming errors early.

Many-to-many relationships

In our forum example, we need moderators for forums. Each forum can have many moderators (users), and each moderator can be a moderator of many forums.



In this case we use a **junction table** (aka **join table**). If our forum and user tables looked like this:

User

id	username	password
1	barry	******
2	bob	*****

Forum

id	name
1	Forum A
2	Forum B

Let's says Barry is a mod of Forum A, and Bob is a mod of Forum A and Forum B. Our join table, typically called user_forum, would look like this:

user_id	forum_id
1	1
2	1
2	2

Using this we can store information about which users are moderators of which forums.

SQL Join Queries

Our many-to-many table (user_forum) would usually take a few queries to join up all the data from the “user” and the “forum” tables.

JOIN queries are a way of of concatenating tables based on matching values, i.e. of turning

user_id	forum_id
1	1
1	2
2	1

into

forum_id	name	user_id	user	password
1	Forum A	1	admin	d033e22ae348aeb5660fc2140aec35850c4da997
1	Forum A	2	barry	031baaf2498d86a0787e99508ff3317ad7871b06

Inner Join

Here's an example of a many-to-many join query:

```
Show all columns from the user and forum tables
SELECT user.*, forum.* FROM `user_forum`
Join rows from the “forum” and “user” tables where the ID is the same
JOIN forum ON forum.id = user_forum.forum_id
JOIN user ON user.id = user_forum.user_id
Show only rows where user_id is 1
WHERE user_id = 1
```

For info on other types of Join queries, see “CodingHorror-A visual explanation of SQL joins” link in wiki