1 Lezione del 02-12-24

1.1 Connettere PHP a MySQL

Vediamo come connetterci, attraverso script PHP, al database MySQL (MariaDB su LAMP). Ciò che vogliamo fare è essenzialmente:

- Connetterci al DBMS;
- Gestire eventuali errori di connessione:
- Eseguire query sul database;
- Elaborare i risultati;
- Liberare le risorse e chiudere la connessione.

Esistono più librerie per l'interfacciamento con DBMS:

- **mysqli**, procedurale o orientata agli oggetti;
- **PDO**, orientata agli oggetti.

1.1.1 Connettersi al database

Vediamo come stabilire connessioni al database con mysqli e PDO:

- **mysqli**:

```
// queste variabili variano di installazione in installazione
$host = "localhost";
$database = "testdb";
$user = "testuser";
$pass = "testpassword";

$connection = my_sqli_connect($host, $user, $pass, $database);
```

- **PDO**:

```
// queste variabili variano di installazione in installazione
$connection_string = "mysql:host=localhost;dbname=testdb";
$user = "testuser";
$pass = "testpassword";

$PDO = new PDO($connection_string, $user, $pass); // questa e' un'
    istanza di classe!
```

Notiamo che scrivere direttamente nel codice di connessione i dettagli del database e dell'utente che vi accede non è molto saggio. Un'opzione migliore è quella di definire variabili apposite:

```
<?php
define("DBHOST", "localhost");
define("DBNAME", "testdb");
define("DBUSER", "testuser");
define("DBPASS", "testpassword");
?>
```

1.1.2 Gestione degli errori

Le due librerie gestiscono diversamente gli errori di connessione:

- **mysqli** usa statement condizionali (if-else) sugli oggetti restituiti dai tentativi di connessione;
- **PDO** si basa su blocchi try-catch di gestione delle eccezioni.

1.1.3 Eseguire query

Vediamo come eseguire query negli oggetti restituiti da mysqli e PDO:

- **mysqli**:

```
$sql = "SELECT * FROM Biglietti WHERE Name = "Rossi"
$result = mysqli_query($connection, $sql);
```

- **PDO**:

```
$sql = "SELECT * FROM Biglietti WHERE Name = "Rossi"
$result = $PDO->query($sql);
```

1.1.4 SQL injection

Una grande falla di sicurezza che possiamo creare quando integriamo contenuti inseriti dall'utente con le nostre query è l'**SQL injection**. Se inseriamo ciò che otteniamo dall'utente così com'è nelle stringhe di query, questi potrebbe **chiudere** lo statement SQL corrente ed avviarne un altro, con conseguenze catastrofiche. Ad esempio, in una casella per l'ingresso dell'username, l'utente potrebbe scrivere:

```
"; TRUNCATE TABLE Users; #
```

A questo punto, se la nostra query aveva l'aspetto:

```
SELECT *
FROM Users
WHERE uname="$user_name"
  AND passwd=MD5("abcd")
```

otterremo la query "maligna":

```
SELECT *
FROM Users
WHERE uname="";
TRUNCATE TABLE Users; # AND passwd=MD5("abcd")
```

Questo comporterebbe l'eliminazione di tutti gli utenti, che chiaramente non è ciò che vogliamo.

Facciamo un'altra considerazione: nel codice presentato, della password viene memorizzato l'hash MD5, e non la password stessa *in chiaro*. Questa è sempre una misura di sicurezza, anche se oggi il semplice hashing MD5 non è più considerato un meccanismo di oscuramento delle password sicuro.

Possiamo proteggerci dall'SQL injection attraverso due meccanismi:

- **Sanitizzazione** dell'input dall'utente: questa si può fare attraverso le funzioni `mysqli_real_escape_stri` `()` in mysqli o `quote()` in PDO;

- L'uso di **prepared statement**: un tipo di query che viene "precompilato" dal DBMS, e su cui la sanificazione viene eseguita automaticamente:
  - **mysqli**:

```php
$id = $_GET["id"];
$sql = "SELECT * FROM Libri WHERE ID=?";

if($statement = mysqli_prepare($connection, $sql)) {
  // tipi di bind: s - stringa, b - blob, i - int, ecc...
  mysqli_stmt_bind_param($statement, "i", $id);
  mysqli_stmt_execute($statement);
}
```

  - **PDO**:

```php
$id = $_GET["id"];

$sql = "SELECT * FROM Libri WHERE ID=?";
$statement = $PDO->prepare($sql);
$statement->bindValue(1, $id);
$statement->execute();
```

1.1.5 Elaborare i result set

Vediamo poi come elaborare i risultati ottenuti dalle query:

- **mysqli**: Abbiamo due alternative a seconda del tipo di query (normale o prepared):
  - **Query standard**:

```php
$sql = "SELECT * FROM Iscrizioni WHERE YEAR(Data) > 2004";

if($result = mysqli_query($connection, $sql)) {
  while($row = mysqli_fetch_assoc($result)) {
    // $row contiene il record corrente
  }
}
```

  - **Query prepared**:

```php
$sql = "SELECT Nome, Data FROM Iscrizioni WHERE YEAR(Data) > 2004
    ";

if($statement = $mysqli_prepare($connection, $sql)) {
  mysqli_stmt_bindm($statement, "i", $id);
  mysqli_stmt_execute($statement);

  mysqli_stmt_bind_result($statement, $name, $data);

  while(mysqli_stmt_fetch($statement)) {
    // $name e $data contentgono i rispettivi attributi del
    record corrente
  }
}

```

- **PDO**:

```php
$sql = "SELECT Nome, Data FROM Iscrizioni WHERE YEAR(Data) > 2004";

$result = $pdo->query($sql);
while($row = $result->fetch()) {
  // $row contiene il record corrente
}
```

Notiamo che attraverso la libreria PDO è possibile estrarre gli attributi di un record in un oggetto anziché un array, a patto che i campi dell'oggetto corrispondano ai nomi degli attributi:

```php
class Libro {
  public $id;
  public $titolo;
  public $anno;
  public $descrizione;
}

// ...

$id = $_GET["ID"];
$sql = "SELECT id, titolo, anno, descrizione FROM Libro WHERE id=?"
$stmt = $pdo->prepare($sql);
$stmt->bindValue(1, $id);
$statement->execute();

$libro = $statement->fetchObject("Libro");
// adesso $book contiene gli attributi del record:
// $libro->id, $libro->titolo, ecc...
```

1.1.6 Chiudere la connessione

Vediamo infine come chiudere le connessioni col DBMS e quindi liberare le risorse relative:

- **mysqli**:

```php
$mysqli_free_result($result);
$mysqli_close($connection);
```

- **PDO**:

```php
$pdo = null;
```

1.1.7 Transazioni

Sia mysqli che PDO supportano il meccanismo delle transazioni:

- **mysqli**:

```php
mysqli_autocommit($connection, FALSE); // di default vale TRUE

// query

if($condition) {
  mysqli_commit($connection);
} else {
  mysqli_rollback($connection);
}
```

- **PDO**:

```
$pdo->setAttribute(PDO::ATTR_ERRMODE, PDO__ERRMODE_EXCEPTION); // per ottenere eccezioni dal DBMS

try {
  $pdo->beginTransaction();

  // query

  $pdo->commit();
} catch (Exception $e) {
  $pdo->rollback();
}
```

1.2 File nel database

Esistono due modi di immagazzinare file in un database:

1. Memorizzare la locazione del file nel filsystem del database;
2. Memorizzare il file direttamente nel database sotto forma di un **blob**. In particolare, vediamo un esempio di codice:

```
$fileContent = file_get_contents("file.rtf");
$sql = "INSERT INTO Documenti (File) VALUES(":data")";

$stmt = $pdo->prepare($sql);
$stmt->bindParam(":data", $fileContent, PDO::PARAM_LOB);
$stmt->execute();

// il file e' nel server, possiamo recuperarlo con:

$sql = "SELECT * FROM Documents WHERE id=:id";
$stmt = $pdo->prepare($sql);
$stmt->bindParam(":id", $_GET["id"]);
$stmt->execute();

$result = $stmt->fetch(PDO::FETCH_ASSOC);
if($result) {
  // il file e' in $result["File"]
}
```

1.3 Gestione dello stato

Gestire lo stato nelle applicazioni web ha delle differenze sostanziali dal farlo nelle applicazioni desktop: mentre quest'ultime hanno a disposizione la memoria del dispositivo dell'utente, le applicazioni web dispongono solo della memoria del server, e quindi non hanno di norma modo di memorizzare informazioni relative alla sessione di un qualsiasi utente.

Possiamo introdurre meccanismi di recupero dello stato, in particolare:

- Stringhe di query: le stesse dei metodi POST e GET, ecc... poco usate per la gestione dello stato;
- **Cookie**.

1.3.1 Cookie

I cookie sono coppie nome-valore memorizzate in file di testo gestiti dal browser (con limite di dimensioni di 4 Kilobyte). Come sempre, visto che sono risorse generate sul lato client, non dobbiamo fidarci ciecamente di ciò che otteniamo dai cookie in quanto potrebbero essere stati manipolati.

Il principio di funzionamento dei cookie è il seguente: quando l'utente visita una pagina, il web server inserisce nell'header della risposta HTTP la rchiesta di impostare determinati cookie. Il browser, ricevute tali richieste, si occupa di impostare i cooke richiesti. Di lì in poi, ad ogni accesso al sito, il browser invierà (idealmente) i cookie che il web server aveva impostato indietro al web server stesso, così che questo possa recuperare informazioni sullo stato.

Le stringhe per l'impostazione di cookie nell'header hanno la forma seguente:

```
Set-Cookie name=value[;expires=date][;domain=domain][;path=path][;secure]
```

Il browser, di contro, invia indietro stringhe del tipo:

```
Cookie name=value
```

PHP fornisce la funzione `setcookie()` per impostare i cookie. Questa va eseguita prima di generare qualsiasi output HTML dallo script.

Vediamo le informazioni che può immagazzinare un cookie:

- **Scadenza** *(Expire)*: indica il tempo di validità del cookie, cioè il tempo per cui dovrebbe continuare ad essere inviato al web server per ogni richiesta HTTP. Un cookie con scadenza 0 è detto **cookie di sessione** e vale solo per la sessione corrente (sono di questo tipo i cookie inviati di default dalla `setcookie()`);
- **Dominio** *(Domain)*: indica il dominio per cui il cookie dovrebbe essere inviato, e può essere anche più grande del dominio corrente.
- **Percorso** *(Path)*: indica il percorso URL che deve esistere nella risorsa richiesta per l'invio del cookie;
- **Cookie sicuri** *(Secure)*: un flag che indica che l'invio del cookie deve accadere solo attraverso richieste SSL col protocollo HTTPS. I cookie inviati su HTTPS sono di default di questo tipo.

1.3.2 Serializzazione

La serializzazione è il processo attraverso il cui si trasformano oggetti arbitrariamente complicati in sequenze di *bit*, *byte*, o a livelli di astrazione superiori *caratteri* (solitamente in codifica ASCII o UTF-8).

PHP fornisce le funzioni `serialize()` e `unserialize()` proprio per questi scopi.

1.3.3 Gestione della sessione in PHP

PHP fornisce un meccanismo per la gestione della sessione. Questo si attiva attraverso la funzione `session_start()`. Dalla chiamata di questa funzione in poi, si può usare la variabile superglobale `$_SESSION` per accedere ad informazioni riguardo alla sessione.

A livello implementativo, il meccanismo di gestione della sessione si basa su un **cookie di sessione** da 32 bit che viene trasmesso avanti e indietro fra il client e il server. Al cookie sono associati dati serializzati (sostanzialmente array associativi), memorizzati

nella memoria del server, che vengono restituiti da un componente detto **session state provider**.

Il problema può porsi quando si hanno più web server che si dividono il carico delle richieste: in questo caso i dati relativi ad una sessione potrebbero essere su uno solo di questi server, e un secondo non potrebbe proseguire correttamente la sessione di un utente. Le soluzioni al problema sono le seguenti:

- **Load balancer session-aware**: cioè load balancer che inviano richieste successive sempre allo stesso server in modo da preservare le sessioni in corso. Questo approccio rende il load balancer più complesso e meno efficiente.
- **Server di sessione condiviso**: un server a parte che si occupa di mantenere le informazioni riguardo a tutte le sessioni in corso.

1.3.4 Web storage

Il **web storage** è un meccanismo implementato lato client dal linguaggio JavaScript che permette la di supplementare i cokie attraverso dati memorizzati in locale sulla macchina del client.

Esistono due oggetti di web storage:

- **localStorage**: per informazioni persistenti fra sessioni;
- **sessionStorage**: per informazioni relative a singole sessioni.

Questi oggetti vengono usati come qualsiasi altro oggetto in JavaScript, cioè possono essere forniti di attributi arbitrari.