PDO: PHP DATA OBJECT

PDO, acronimo di PHP Data Object, è un'estensione di PHP introdotta a partire dalla versione 5.1 del linguaggio con lo scopo di unificare le API di accesso ai database. Alla base della loro concezione vi era quindi l'idea di fornire un'unica interfaccia di programmazione con cui interagire con tutte le basi di dati.

Con la versione 7 di PHP, e le successive implementazioni di questa milestone, PDO si avvia a diventare uno standard quanto meno de facto, in particolare perché con questa release è stato rimosso il supporto ormai deprecato alle originarie funzioni specifiche per MySQL (lasciando però disponibile mysqli). Con l'abbandono delle "storiche" mysql functions il passaggio a PDO diventa quindi essenziale per la creazione di applicazioni aggiornate agli standard di sviluppo e sicurezza più recenti.

PDO e la programmazione ad oggetti

Come ben sintetizza il suo acronimo, PDO si presenta come una soluzione per sua natura orientata agli oggetti, l'estensione fornisce infatti un'interfaccia comune per la comunicazione con i diversi database. Tale caratteristica risulta estremamente pratica nel caso di una migrazione da un DBMS ad un altro, e permette agli sviluppatori di utilizzare le medesime strutture di programmazione a prescindere dal database di riferimento.

Quello che non offre PDO è però una funzionalità di traduzione delle query o di emulazione di eventuali funzioni mancanti, sarà quindi compito dello sviluppatore farsi carico delle differenti versioni dei dialetti SQL. PDO non mette infatti a disposizione una vera e propria "database abstraction", ma più semplicemente un data-access abstraction layer, cioè un livello di astrazione per l'accesso ai dati.

I vantaggi nell'uso di PDO

Uno dei principali punti di forza di PDO risiede nel fatto che esso mette a disposizione delle sviluppatore i **prepared statement**, costrutti che se utilizzati sistematicamente sono in grado di mettere al riparo gli script delle minacce basate sull'SQL injection, una tecnica di attacco incentrata sull'iniezione di SQL malevolo tramite input al fine di violare le applicazioni per la gestione dati.

L'interfaccia PDO permette l'accesso, tramite gli opportuni driver, ad alcuni dei database più diffusi, tra di essi vi sono anche:

Firebird

IBM DB2

IBM Informix Dynamic Server

MySQL (versioni supportate 3.x/4.x/5.x)

Oracle

PostgreSQL

SQLite 3 e SQLite 2

Microsoft SQL Server / SQL Azure.

In conclusione è possibile sottolineare che i vantaggi derivanti dall'uso di PDO fanno riferimento alle seguenti caratteristiche:

Portabilità: grazie al supporto di più engine per la gestione di dati.

Programmazione orientata agli oggetti: grazie ad un'interfaccia nativamente object-oriented.

Sicurezza: grazie ai prepared statement che proteggono dai tentativi di code injection.

L'uso di PDO rappresenta quindi il primo passo per rendere la nostra applicazione object-oriented e riusabile.

Connessione a MySql con PDO

Impariamo creare delle connessioni, anche persistenti, ad un database MySQL tramite l'istanza di un oggetto PDO in PHP.

Istanza della classe PDO

La connessione a un database MySQL si realizza creando un'istanza della classe PDO: il costruttore si aspetta 3 parametri, uno obbligatorio e due facoltativi. Il parametro obbligatorio è il DSN (Data Source Name), i parametri facoltativi, almeno formalmente, sono username e password di accesso al nostro database server.

Supponendo che il database server sia in locale e il database al quale vogliamo accedere si chiami "corso" il codice necessario alla connessione sarà il seguente:

```php
$hostname = "localhost";
$dbname = "corso";
$user = "nome_utente";
$pass = "password";
$db = new PDO ("mysql:host=$hostname;dbname=$dbname", $user, $pass);
```

La variabile $user conterrà lo username di accesso e la variabile password la relativa password di autenticazione.

Possiamo inserire la creazione della connessione al database in un costrutto try-catch in modo da gestire eventuali errori (PDOException) di connessione, bloccare l'esecuzione del codice se la connessione stessa non va a buon fine e visualizzare la tipologia di eccezione individuata ($e->getMessage()):

```php
try {
    $hostname = "localhost";
    $dbname = "corso";
    $user = "nome_utente";
    $pass = "password";
    $db = new PDO ("mysql:host=$hostname;dbname=$dbname", $user, $pass);
} catch (PDOException $e) {
    echo "Errore: " . $e->getMessage();
    die();
}
```

Se viceversa la connessione andasse a buon fine avremmo a disposizione un'istanza dell'oggetto PDO.

Creazione di una connessione persistente

In alcune circostanze potrebbe essere opportuno avere a disposizione delle connessioni persistenti, ovvero delle connessioni che non vengono distrutte al termine dell'esecuzione dello script ma memorizzate in cache per poter essere riutilizzate da un altro script che impiega le medesime credenziali. Lo scopo è quello di migliorare le prestazioni dato che lo script non deve riaprire una connessione al database server ogni volta che questa diventa necessaria.

Per creare una connessione persistente dovremo modificare il nostro codice per la connessione in questo modo:

```
$db = new PDO("mysql:host=$hostname;dbname=$dbname", $user, $pass, array(
    PDO::ATTR_PERSISTENT => true
));
```

PDO: operazioni di scrittura sul database

Impariamo ad utilizzare i metodi di PDO per scrivere dati in un database MySQL con PHP e scopriamo quali accorgimenti adottare per evitare i pericoli della SQL INJECTION.

Ora che abbiamo a disposizione la connessione al database possiamo iniziare a lavorare con i dati, in particolare ci dedicheremo alle operazioni in scrittura su un database. Innanzitutto supponiamo di avere a disposizione un database MySQL al quale connettersi:

```php
try {
    $db = new PDO('mysql:host=localhost;dbname=corso', $user, $pass);
} catch (PDOException $e) {
    echo "Errore: " . $e->getMessage();
    die();
}
```

Nel database corso avremo una tabella utenti dove il campo id è la chiave primaria autoincrementale:

```sql
CREATE TABLE utenti (
  id int(11) NOT NULL,
  nome varchar(255) COLLATE utf8_unicode_ci NOT NULL,
  cognome varchar(255) COLLATE utf8_unicode_ci NOT NULL,
  email varchar(255) COLLATE utf8_unicode_ci NOT NULL,
  anno_nascita year(4) NOT NULL
) ENGINE=MyISAM DEFAULT CHARSET=utf8 COLLATE=utf8_unicode_ci;
```

Inserire record

Il primo punto da affrontare è l'inserimento dei record nella tabella. PDO mette a disposizione diverse possibilità, più o meno "raccomandabili". La prima consiste nel creare una query ed eseguirla direttamente dalla pagina PHP:

```php
$db->query("INSERT INTO utenti(nome, cognome, email) VALUES('lorenzo','de ambrosis','email@test.it','1973')");
```

Soluzione semplice ma in realtà poco praticabile: infatti quasi sempre alle stringhe dovremo sostituire delle variabili:

```php
$db->query("INSERT INTO utenti(nome, cognome, email)
VALUES('$nome','$cognome','$email','$anno')");
```

L'esecuzione diretta della query proposta dell'esempio ci espone però al rischio di SQL INJECTION dato che non c'è nessun controllo sul dato inserito. L'unica possibile opzione è quella di usare il metodo quote che funziona come mysql_real_escape_string(), ma non è la soluzione ottimale.

Sarà quindi opportuno passare alle query parametriche.

Query parametriche

La struttura della nostra query cambierà ora notevolmente, avremo infatti le fasi prepare, il bind dei parametri e infine l'esecuzione della query. Iniziamo quindi a definire l'istruzione SQL:

```
$sql = "INSERT INTO utenti(nome, cognome, email)
VALUES(':nome',':cognome',':email',':anno')");
```

La differenza rispetto all'approccio precedente sta nella sostituzione delle variabili, per esempio $anno viene rappresentato dal parametro :anno. A questo punto possiamo preparare la nostra query:

```
$stmt = $db->prepare($sql);
```

E a questo punto eseguiamo il bind dei parametri:

```
$stmt->bindParam(':nome', $nome, PDO::PARAM_STR);
```

Il metodo bindParam accetta 3 argomenti, il nome del parametro, il valore del parametro e infine il tipo di dato, parametro opzionale.

Infine, ultimo step, introduciamo il metodo execute:

```
$stmt->execute();
```

L'istruzione SQL e i parametri vengono spediti separatamente al database server (motivo per il quale non sarà possibile stampare la query "completa") e solo dopo la query verrà finalmente eseguita. Questo modo di procedere è quello che ci fornisce una maggiore sicurezza rispetto ai pericoli derivanti dall'iniezione di codice SQL.

bindParam vs. bindValue

Un approfondimento necessario riguarda la differenza fra bindParam e bindValue: bindValue associa il valore della variabile al parametro, bindParam referenzia invece la variabile agganciandola al parametro, in poche parole con bindValue non possiamo modificare il valore associato, mentre usando bindParam verrà usato il nuovo valore se quello della variabile cambia tra il bind e l'execute.

Quindi mentre

```php
$sql = "INSERT INTO utenti(nome) VALUES(':nome');
$stmt = $db->prepare($sql);
$nome = 'Lorenzo';
$stmt->bindValue(':nome', $nome, PDO::PARAM_STR);
$stmt->execute();
```

inserirà il valore "Lorenzo"

```php
$sql = "INSERT INTO utenti(nome) VALUES(':nome');
$stmt = $db->prepare($sql);
$nome = 'Lorenzo';
$stmt->bindParam(':nome', $nome, PDO::PARAM_STR);
$nome = 'Pippo';
$stmt->execute();
```

inserirà il valore "Pippo".

PDO: estrazione dati dal database

Affrontiamo ora l'argomento riguardante l'estrazione dei dati da un database e della visualizzazione degli stessi in una pagina Web. Come sempre, supponiamo di avere a disposizione la connessione al nostro database e prepariamo la prima query con lo schema proposto nella lezione precedente.

```
$sql = 'SELECT nome, cognome, FROM utenti';
$stmt = $db->prepare($sql);
$stmt->execute();
```

Un'ulteriore questione, abbastanza frequente in ambito pratico, è legata all'uso di SELECT con un LIKE nella clausola WHERE combinando il tutto con una query parametrica. In questo caso la parte che rappresenta il parametro sarà solo quella della variabile, senza tenere in considerazione i caratteri jolly che fanno parte dell'SQL:

```
$sql = 'SELECT nome, cognome, FROM utenti WHERE email like "%:email%"';
$stmt = $db->prepare($sql);
$stmt->bindParam(':email', $email, PDO::FETCH_ASSOC);
$stmt->execute();
```

A questo punto, un primo problema interessante potrebbe essere questo: sono stati estratti record? E se sì, quanti? Per rispondere a questa domanda possiamo memorizzare il numero totale di record estratti in una variabile totale e usare il metodo rowCount():

```
$totale = $stmt->rowCount();
```

L'istruzione fetch

Supponendo che siano stati estratti dei record dovremo eseguire un ciclo per stampare tali risultati nella nostra pagina, effettueremo tale procedura nello stesso modo in cui opereremmo per compiere qualsiasi altra operazione legata all'uso di dati estratti dal database:

```
while($row = $stmt->fetch(PDO::FETCH_ASSOC)){
    echo '
' . $row['nome'] . ' ' . $row['cognome'] . '
';
}
```

Con questa istruzione facciamo scorrere i dati estratti dal database associando ad ogni riga la variabile $row grazie al metodo fetch(), invece PDO::FETCH_ASSOC ci dice che stiamo creando un array associativo usando come chiavi i nomi dei campi, questa variabile è più utilizzata rispetto a PDO::FETCH_NUM che usa un indice numerico e rende quindi obbligatorio rispettare l'ordine dei campi espresso nell'istruzione SELECT.

L'istruzione fetchAll

Un caso abbastanza interessante è costituito dalla situazione in cui vogliamo inviare al client l'output in formato JSON, in questo caso non è necessario eseguire il loop con while o foreach e, più semplicemente, possiamo usare l'istruzione fetchAll. Quindi dopo l'esecuzione della query il codice necessario potrebbe essere:

```php
$rows = $stmt->fetchAll(PDO::FETCH_ASSOC);
print json_encode($rows);
```

Quest'ultimo caso si presta anche ad una variante di utilizzo dati con un ciclo foreach:

```php
foreach($rows as $row){
echo '
' . $row['nome'] . ' ' . $row['cognome'] . '
';
}
```

Da un punto di vista tecnico fetchAll è più veloce rispetto al semplice fetch ma consuma un maggiore quantitativo di memoria.