

## VII - Manipulation de données

Dans un environnement Web, il y a trois sources principales de données : une base de données relationnelle ou objet, des fichiers XML et des services Web.


### VII-A - Bases de données relationnelles

#### VII-A-1 - Introduction

Les bases de données permettent d'emmagasiner et de retrouver de manière simple et efficace de très grandes quantités d'informations de tous types : numérique, texte, multimédia...

Pour accéder à une base de données, il faut généralement disposer d'un code d'accès, de l'adresse IP du serveur et du nom de la base de données sur le serveur. Une dernière information nous indique le type de la BDD qui nous servira pour lancer nos requêtes (MySQL, PostgreSQL, Oracle...).

La plupart des APIs fonctionnent sur un principe procédural, orienté objet ou bien les deux. Nous utiliserons de préférence la POO pour la flexibilité qu'elle nous offre.

 *Je vais partir du principe que vous connaissez les bases de données relationnelles et le langage SQL.*

L'accès à une BDD peut se faire soit au moyen d'une API spécialisée pour votre BDD (MySQL, PostgreSQL, Oracle...), soit à travers une API générique (aussi appelée **couche d'abstraction**). Puisque les couches d'abstraction laissent davantage de liberté et de flexibilité que les APIs, il n'y a aucun avantage à utiliser une API spécifique. En revanche, le plus gros avantage d'utiliser une telle couche d'abstraction est de donner le choix du SGBD avec un minimum de modifications sur le code de l'application PHP.

Je vous propose donc d'utiliser **PHP Data Objects** (alias **PDO**), qui est la couche d'abstraction fournie en standard dans PHP.

Extensions PHP nécessaires : **php\_pdo** + **php\_pdo\_mysql** (cette deuxième extension correspond à votre SGBD, dans mon cas MySQL).

L'avantage principal de PDO est l'extrême simplicité de changement de SGBD : il suffit de changer un paramètre lors de l'instanciation de PDO, le reste du code est inchangé. Cela permet par exemple de coder une application et de la vendre à n'importe quel client, quel que soit son SGBD de prédilection, sans surcharge de travail. Par conséquent, il n'est pas nécessaire d'apprendre le fonctionnement de chaque API spécifique, PDO est suffisant. Vos développements prennent donc moins de temps.

 *Des exemples de chaînes de connexion se trouvent  dans la FAQ PHP, et d'autres tutoriels sont disponibles ici :  Cours SGBD en PHP.*

#### VII-A-2 - Accès aux données

**L'accès à une base de données est soumis à diverses étapes :**

- Ouverture de la connexion au serveur ;
- Choix de la BDD sur le serveur (cela se fait parfois au moment de l'ouverture de la connexion) ;
- Préparation de requêtes (facultatif, dépend de l'API utilisée) ;
- Exécution de requêtes et récupération des résultats.

L'envoi de requêtes implique souvent le parcours des résultats, et dans certains cas la préparation préalable des requêtes (c'est le cas de PDO).

La connexion au serveur et la sélection de la BDD se font lors de l'instanciation de l'objet PDO :

```
$db = new PDO('mysql:host=localhost;dbname=developpez', 'utilisateur', 'motdepasse');
```

Pour envoyer une requête, il y a plusieurs solutions. La plus sécurisée est de passer par des requêtes préparées (*prepared statements*) :

```
$db = new PDO('mysql:host=localhost;dbname=developpez', 'utilisateur', 'motdepasse');

$select_users = $db->prepare('SELECT id, name FROM user');
$select_user = $db->prepare('SELECT id, name FROM user WHERE user_id = ?');
$insert_user = $db->prepare('INSERT INTO user (name, password) VALUES (?, ?)');

//récupère tous les utilisateurs (aucun puisque la table est vide)
$select_users->execute();
$users = $select_users->fetchAll();

$insert_user->execute(array('BrYs', '4321'));
$insert_user->execute(array('mathieu', '4321'));
$insert_user->execute(array('Yogui', '4321'));


$select_users->execute(); //récupère tous les utilisateurs (les 3)
$users = $select_users->fetchAll();

$select_user->execute(array(1)); //récupère l'utilisateur numéro 1
$user_1 = $select_user->fetchAll();

$select_user->execute(array(2)); //récupère l'utilisateur numéro 2
$user_2 = $select_user->fetchAll();

$select_user->execute(array(3)); //récupère l'utilisateur numéro 3
$user_3 = $select_user->fetchAll();
```

Comme nous pouvons le voir, une requête se prépare une seule fois pour l'ensemble du script. Chaque exécution de la requête se fait alors par la méthode **execute()**, et la récupération des résultats par la méthode **fetchAll()**.

Bien sûr, il est possible d'envoyer des requêtes directement depuis l'objet **\$db**, mais alors il n'y a aucune forme de sécurisation des données puisque la méthode **quote()** n'est pas disponible pour tous les pilotes de BDD. Un code qui envoie des requêtes non préparées n'est donc pas  **portable**.

## Il y a deux avantages majeurs à préparer les requêtes :

- Chaque exécution est plus rapide en requête préparée qu'en envoyant la totalité de la requête à chaque fois ;
- Les paramètres sont transmis à la requête sous forme binaire, ce qui évite tout risque d'injection SQL.

Les paramètres peuvent être transmis à la requête préparée soit dans l'ordre, soit sous forme nommée. Les paramètres nommés peuvent être envoyés dans n'importe quel ordre :

```
<?php
$db = new PDO(
    'mysql:host=localhost;dbname=developpez',
    'utilisateur',
    'motdepasse');

$insert_1 = $db->prepare(
    'INSERT INTO user (name, password) VALUES (?, ?)');

$insert_2 = $db->prepare(
```

```
'INSERT INTO user (name, password) VALUES (:name, :password)');

$insert_1->execute(array('BrYs', '4321'));
$insert_2->execute(array(
    ':password' => '4321',
    ':name' => 'BrYs'
));
```

## VII-A-3 - Performances

Dans une architecture classique, le serveur HTTP et le SGBD sont deux machines différentes sur un même réseau local. Exactement combien de machines entrent en jeu n'est pas notre propos, mais cela nous indique le protocole de communication habituellement utilisé pour le dialogue entre les deux applications (par exemple entre Apache et MySQL) : TCP/IP. Toutes les requêtes et leurs résultats transitent par la couche réseau. Il est donc plutôt facile de congestionner un réseau avec des requêtes renvoyant de nombreux résultats, surtout si elles sont fréquemment exécutées. C'est l'une des raisons en défaveur de la pratique "SELECT \*", puisqu'elle retourne plus de champs qu'il n'est nécessaire, et fait donc transiter un grand nombre d'informations inutiles par le réseau.

C'est aussi à cause de ces échanges qu'il est important de réduire le nombre de requêtes exécutées à chaque page, sans pour autant que cela devienne une guerre sainte... Chaque exécution de requête implique d'attendre la réponse du SGBD avant de pouvoir poursuivre l'exécution du script, ce qui correspond à un aller-retour TCP/IP entre le serveur HTTP et le SGBD. La communication entre deux machines par un réseau local est certes rapide, mais c'est sans commune mesure avec des optimisations de type *"for plutôt que foreach"*. Soyez conscients que ce qui ralentit le plus votre application est sans doute davantage le temps nécessaire pour faire un aller-retour de requête et de son résultat, que des optimisations in-line de votre code PHP. J'ai entendu **Tim Bray** rappeler aux développeurs qu'un système de fichiers est souvent plus rapide qu'un SGBD : évaluez votre besoin, comparez ce que vous apporte chaque SGBD. SQLite est parfois un bon compromis.

Le principe des requêtes préparées permet d'enregistrer temporairement chaque requête du côté du SGBD. Ainsi, en plus des optimisations que cela permet d'obtenir côté serveur, pour exécuter chaque requête il suffit de renvoyer les paramètres (au lieu de l'ensemble de la requête). Avec l'avantage sécurité que nous avons déjà mentionné, cela nous donne une longue liste d'avantages en faveur des requêtes préparées...

À des fins d'illustration, voici un exemple (à ne pas suivre) utilisant une base de données de forum avec une clef étrangère entre **message.user\_id** et **user.id** :

```
<?php
mysql_connect('localhost', 'utilisateur', 'motdepasse');
mysql_select_db('developpez');

$sql = 'SELECT id, title, user_id
      FROM message
      ORDER BY post_date';
$db_messages = mysql_query($sql);

header('Content-Type: text/html; charset=utf-8');
while($message = mysql_fetch_array($db_messages))
{
    $sql = 'SELECT name
          FROM user
          WHERE id = ' . (int)$message['user_id'];

    $db_user = mysql_query($sql);
    $user = mysql_fetch_array($db_user);

    echo utf8_encode(
        htmlspecialchars($message['title'].' par '.$user['name'], ENT_QUOTES)
        . '<br/>';
}
```

## Voici les erreurs du script ci-dessus :

- Utilisation de l'API MySQL plutôt que **PDO** ou même **MySQLi**, ce qui est un mauvais choix par rapport aux performances et à la portabilité du code ;
- Utilisation de **\*fetch\_array**, qui retourne un tableau de résultats à la fois sous forme d'index numérique et d'index associatif, ce qui consomme inutilement de la mémoire ;
- Exécution de nombreuses requêtes alors qu'**INNER JOIN** aurait été plus efficace.

Le script suivant donne le même résultat :

```
<?php
$db = new PDO(
    'mysql:host=localhost;dbname=developpez',
    'utilisateur',
    'motdepasse');

$sql = 'SELECT m.id, m.title, u.name AS author
FROM message AS m
INNER JOIN user AS u ON m.user_id = u.id
ORDER BY post_date';

$select_messages = $db->prepare($sql);
$select_messages->setFetchMode(PDO::FETCH_ASSOC);
$select_messages->execute();

header('Content-Type: text/html; charset=utf-8');
foreach($select_messages->fetchAll() as $message)
{
    echo utf8_encode(htmlspecialchars(
        $message['title'].' par '.$message['author'], ENT_QUOTES))
        . '<br/>';
}
```

## VII-A-4 - Bonnes pratiques

La POO autorisant le principe d'héritage, autant en profiter. Dans l'exemple ci-dessus, on peut voir que les sélections utilisent deux méthodes à la suite (**execute()** puis **fetchAll()**) : cela peut être simplifié par l'héritage. Il faut également prendre en compte la configuration de la classe : chaque connexion dispose d'un certain nombre d'attributs, et PDO peut être configuré pour envoyer des exceptions plutôt que des erreurs PHP. Tout cela peut être fait dans une classe dérivée de PDO, afin de nous assurer que tous nos objets **\$db** en bénéficient dans tous nos projets.

Nous aimerions avoir du code applicatif simple et concis, par exemple :

```
<?php
$db = new MyPDO(
    'mysql:host=localhost;dbname=developpez', 'utilisateur', 'motdepasse');

$userTable = new UserTable($db);

$userTable->truncate();
$userTable->insert('Yogui', '4321');
$userTable->insert('mathieu', '4321');
$userTable->insert('BrYs', '4321');

?><pre><?php
print_r($userTable->selectById(1));
print_r($userTable->selectByName('u'));
print_r($userTable->selectAll());
```

Pour y parvenir, nous avons besoin d'une classe "UserTable" qui représente la table du même nom :

```
<?php
class UserTable
```

```

{
    /*
     * Requêtes préparées
     */
    private $selectById;
    private $selectByName;
    private $selectAll;
    private $insert;
    private $truncate;

    public function __construct($db)
    {
        $this->selectAll = $db->prepare(
            "SELECT id, name FROM user ORDER BY name, id");

        $this->selectById = $db->prepare(
            "SELECT id, name FROM user WHERE id = ?");

        $this->selectByName = $db->prepare(
            "SELECT id, name FROM user WHERE name LIKE ? ORDER BY name, id");

        $this->insert = $db->prepare(
            "INSERT INTO user (name, password) VALUES (:name, :password)");

        $this->truncate = $db->prepare("TRUNCATE user");
    }

    public function insert($name, $password)
    {
        $this->insert->execute(
            array(':name' => $name, ':password' => $password));

        return $this->insert->rowCount();
    }

    public function selectAll()
    {
        $this->selectAll->execute();
        return $this->selectAll->fetchAll();
    }

    public function selectById($id)
    {
        $this->selectById->execute(array($id));
        return $this->selectById->fetch();
    }

    public function selectByName($name)
    {
        $this->selectByName->execute(array('%'.$name.'%'));
        return $this->selectByName->fetchAll();
    }

    public function truncate()
    {
        return $this->truncate->execute();
    }
}

```

Tout cela repose sur notre classe dérivée de PDO, qui configure la connexion ainsi que la manière de préparer les requêtes :

```

<?php
class MyPDO extends PDO
{
    public function __construct($dsn, $user=NULL, $password=NULL)
    {
        parent::__construct($dsn, $user, $password);
    }
}

```

```

        $this->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
    }

    public function prepare($sql, $options=NULL)
    {
        $statement = parent::prepare($sql);
        if(strpos(strtoupper($sql), 'SELECT') === 0) //requête "SELECT"
        {
            $statement->setFetchMode(PDO::FETCH_ASSOC);
        }

        return $statement;
    }
}

```

Résultat :

```

Array
(
    [id] => 1
    [name] => Yogui
)
Array
(
    [0] => Array
        (
            [id] => 2
            [name] => mathieu
        )

    [1] => Array
        (
            [id] => 1
            [name] => Yogui
        )
)
Array
(
    [0] => Array
        (
            [id] => 3
            [name] => BrYs
        )

    [1] => Array
        (
            [id] => 2
            [name] => mathieu
        )

    [2] => Array
        (
            [id] => 1
            [name] => Yogui
        )
)


```



Maintenant est un bon moment pour vous remémorer mon exemple de code lors de l'explication des Late Static Bindings, et pour faire des essais de votre côté.

Tous les frameworks PHP proposent une solution bien mieux adaptée que celle que je présente ici. Par exemple, symfony génère deux classes pour chaque table de la BDD : l'une contient uniquement des méthodes statiques d'accès aux données au niveau de la table, et l'autre est la représentation objet d'un seul tuple

de la table. Zend Framework, quant à lui, opte pour deux classes à dériver : Zend\_Db\_Table\_Abstract et Zend\_Db\_Table\_Row\_Abstract.

 **La base de données étant une source de données externe au script, les données qui en proviennent doivent être traitées de la même manière que les données utilisateur : avec prudence. Il faut valider et filtrer les données issues de la BDD.**

Pour information, voici le code applicatif ci-dessus écrit avec l'API de base de PHP pour MySQL. Le majeur problème de cette approche est qu'elle est spécifique à MySQL et que nous obligeons ainsi tous nos clients à utiliser MySQL, même s'ils avaient déjà un SGBD en lequel ils avaient confiance (par exemple PostgreSQL ou Oracle). Changer de SGBD dans nos scripts nous oblige à effectuer de nombreuses modifications : PDO permet d'éviter ce genre de problèmes.

```
<?php
mysql_connect('localhost', 'utilisateur', 'motdepasse');
mysql_select_db('developpez');
```

```
db_user_truncate();
db_user_insert('Yogui', '4321');
db_user_insert('mathieu', '4321');
db_user_insert('BrYs', '4321');
```

```
?><pre><?php
print_r(db_user_select_by_id(1));
print_r(db_user_select_by_name('u'));
print_r(db_user_select_all());
```

```
<?php
function db_user_insert($name, $password)
{
    $sql = "INSERT INTO user(name, password) VALUES('%s', '%s')";
    mysql_query(sprintf(
        $sql,
        mysql_real_escape_string($name),
        mysql_real_escape_string($password)));

    return mysql_affected_rows();
}

function db_user_select_by_id($id)
{
    $sql = "SELECT id, name FROM user WHERE id = %d";
    $db_result = mysql_query(sprintf($sql, $id));
    if($user = mysql_fetch_assoc($db_result))
    {
        return $user;
    }
    else
    {
        return NULL;
    }
}

function db_user_select_by_name($name)
{
    $sql = "SELECT id, name FROM user WHERE name LIKE '%s' ORDER BY name";
    $db_result = mysql_query(sprintf(
        $sql,
        mysql_real_escape_string('%'.$name.'%')));

    $users = array();
    while($user = mysql_fetch_assoc($db_result))
    {
        $users[] = $user;
    }
    return $users;
}
```

```
function db_user_select_all()
{
    $sql = "SELECT id, name FROM user ORDER BY name";
    $db_result = mysql_query($sql);
    $users = array();
    while($user = mysql_fetch_assoc($db_result))
    {
        $users[] = $user;
    }
    return $users;
}

function db_user_truncate()
{
    mysql_query("TRUNCATE user");
}
```

D'autres inconvénients de l'approche par API spécifique est que ces API ne disposent pas toujours de méthodes pour protéger les requêtes, comme ici **mysql\_real\_escape\_string()**. Par ailleurs, il est très facile d'oublier d'utiliser ces fonctions ou de transtyper la valeur. Cela peut très facilement conduire à des failles de sécurité dans les applications.

Il est donc indéniablement préférable d'utiliser des requêtes préparées.



## VII-B - Fichiers XML

### VII-B-1 - Introduction

Extensions nécessaires : aucune, tout est inclus en standard dans PHP.

Les fichiers XML sont très utiles pour faire transiter les informations entre deux bases de données, par e-mail, etc. Naturellement, PHP fournit des classes permettant de faciliter leur manipulation.

#### Il existe deux méthodes d'accès à un fichier XML :

-  **SAX** : Lecture ou écriture séquentielle, comme un flux de données, au fur et à mesure de l'avancement du pointeur de fichier ; cette approche est avantageuse pour la faible quantité de mémoire utilisée, puisque chaque noeud est libéré de la mémoire après utilisation, mais elle est très complexe à utiliser ;
-  **DOM** : Lecture de l'arbre complet du XML lors de l'ouverture du document (ou manipulation par hiérarchie dans le cas d'écriture) ; cette approche est plus simple mais plus gourmande en mémoire, et n'est pas adaptée pour les grands fichiers.

Nous allons nous concentrer sur une approche DOM, suffisante pour débiter et dans de très nombreuses situations.

### VII-B-2 - Lecture : SimpleXML

SimpleXML est l'API la plus simple pour la lecture de documents XML. La lecture du fichier se fait selon une approche DOM, et on peut facilement filtrer les données par des requêtes XPath.

Prenons l'exemple d'une BDD de messages. Le fichier XML suivant peut être le dump d'une table *message* par exemple, et la table *user* peut être dans un autre fichier XML.

```
messages.xml
<?xml version="1.0" encoding="UTF-8" ?>
<messages>
  <message>
```



### messages.xml

```
<id>1</id>
<user_id>1</user_id>
<title>Bonjour</title>
<body>Un bonjour de Paris ;)</body>
</message>
<message>
  <id>2</id>
  <user_id>2</user_id>
  <title>Super site</title>
  <body>J'apprécie ton site, continue ainsi !</body>
</message>
<message>
  <id>3</id>
  <user_id>2</user_id>
  <title>Merci pour tout</title>
  <body>J'oubliais de dire que ton site m'a beaucoup servi !</body>
</message>
</messages>
```

### index.php

```
<?php
//lecture du XML dans un objet PHP
$message = simplexml_load_file('messages.xml');

foreach($messages as $message) //parcours du XML comme d'un tableau
{
    $id = (int)$message->id;

    //pas de validation, le message doit être affiché tel quel
    $body = $message->body;
    echo html($id.' - '.$body).<br/>;
}

function html($string)
{
    return utf8_encode(htmlspecialchars($string, ENT_QUOTES));
}
```

```
1 - Un bonjour de Paris ;)
2 - J'apprécie ton site, continue ainsi !
3 - J'oubliais de dire que ton site m'a beaucoup servi !
```

Notez que j'ai pris soin de filtrer le texte (décoder l'UTF-8) lorsque je le récupère : cela m'évite d'avoir des problèmes d'encodage. Notez également que l'on peut utiliser **foreach** sur un objet **SimpleXMLElement** : c'est parce qu'il implémente l'interface **ArrayAccess** issu de la **SPL**.



*Un fichier XML est à considérer comme toute autre source de données externes au script :  
il faut filtrer les valeurs qui en proviennent.*

## VII-B-3 - Écriture : DOM

Admettons maintenant que nous ayons besoin d'écrire le fichier XML ci-dessus à partir de notre BDD. Voici le SQL pour la mise en place de la table et de son contenu :

```
CREATE TABLE message (
  id int(12) NOT NULL auto_increment,
  user_id int(12) default NULL,
  title varchar(50) default NULL,
  body varchar(1000) default NULL,
  PRIMARY KEY (id)
);

INSERT INTO message (user_id, title, body)
```

```
1,
"Bonjour",
"Un bonjour de Paris ;)");

INSERT INTO message (user_id, title, body)
VALUES (
2,
"Super site",
"J'apprécie ton site, continue ainsi !");

INSERT INTO message (user_id, title, body)
VALUES (
2,
"Merci pour tout",
"J'oubliais de dire que ton site m'a beaucoup servi !");
```

Nous aurons besoin de notre classe MyPDO et d'une classe MessageTable :

```
<?php
class MessageTable
{
    private $select;

    public function __construct($db)
    {
        $this->select = $db->prepare(
            "SELECT id, user_id, title, body FROM message");
    }

    public function selectAll()
    {
        $this->select->execute();
        return $this->select->fetchAll();
    }
}
```

Ajoutons à cela une classe permettant de simplifier la création du XML :

```
<?php
class MyDOMDocument extends DOMDocument
{
    public function __construct($version=NULL, $charset=NULL)
    {
        parent::__construct($version, $charset);
        $this->strictErrorChecking = TRUE;
        $this->appendChild($this->createElement('messages'));
        $this->formatOutput = TRUE;
    }

    public function appendMessage($id, $user_id, $title, $body)
    {
        $message = $this->createElement('message');
        $this->documentElement->appendChild($message);

        $message->appendChild(
            $this->createElement('id', (int)$id));

        $message->appendChild(
            $this->createElement('user_id', (int)$user_id));

        $message->appendChild(
            $this->createElement('title', $this->escapeText($title)));

        $message->appendChild(
            $this->createElement('body', $this->escapeText($body)));
    }
}
```

```

    }

    protected function escapeText($string)
    {
        return utf8_encode(htmlspecialchars($string));
    }
}

```

Et enfin notre script principal :

```

<?php
$db = new PDO(
    'mysql:host=localhost;dbname=developpez', 'utilisateur', 'motdepasse');

$xml = new MyDOMDocument('1.0', 'utf-8');
$messageTable = new MessageTable($db);


foreach($messageTable->selectAll() as $message)
{
    $id = (int)$message['id'];
    $user_id = (int)$message['user_id'];
    $body = preg_replace('/[[:cntrl:]]/', ' ', $message['body']);



    if(ctype_print($message['title']))
    {
        $title = $message['title'];
    }
    else
    {
        $title = preg_replace('/[^[:graph:]]/', ' ', $message['title']);
    }

    $xml->appendMessage($id, $user_id, $title, $body);
}

header('Content-Type: application/xml; charset=utf-8');
echo $xml->saveXML();

```

 Notez comme je filtre les données en provenance de la BDD dans l'appel à **appendMessage()**. De plus, je les convertis selon leur format de destination lors de l'appel à **createElement()**.

 L'objet **\$messageTable** n'ayant aucune raison d'apparaître plusieurs fois dans notre code, on peut mettre en place un système pour éviter la création de multiples instances de la classe **MessageTable**. Il s'agit du  **design pattern** "Singleton", que nous aborderons plus loin.

