2. Pages Web dynamiques et bases de données

Une page Web dynamique est générée automatiquement grâce à l'exécution d'un script (PHP par exemple). C'est le résultat de l'exécution de ce script (code HTML) qui est renvoyé par le serveur au client. Pour générer ce code HTML, le script peut avoir besoin de données qui sont stockées en dehors du script, notamment dans des bases de données. Dans ce cas, lors de l'exécution du script, l'interpréteur PHP interroge la base de données via des requêtes SQL et utilise les informations renvoyées pour générer le code HTML. Le processus de génération de pages Web dynamiques peut être représenté par la figure 2.1.

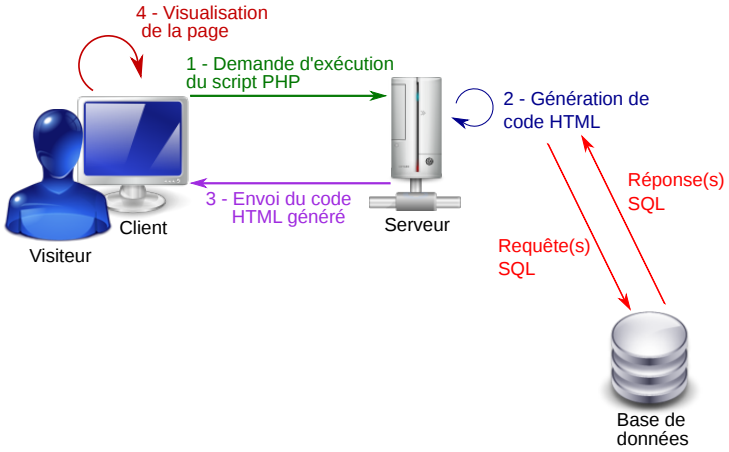

FIGURE 2.1 – Schéma de requête en PHP avec accès à une base de données

R *Lors de l'exécution d'un script, il peut y avoir plusieurs requêtes SQL.*

Les bases de données d'un site Web vont pouvoir stocker des informations utiles pour ce site. La base de données d'un forum va stocker les informations des utilisateurs et tous les messages postés. Pour un site marchand, la base de données contiendra la liste des clients/fournisseurs/marchandises. Dans le cas d'un moteur de recherche, la base contiendra les informations extraites de toutes les pages indexées.

2.1 PHP et bases de données

L'accès aux bases de données dans un script PHP se fait à l'aide de la librairie *PHP Data Objects*[1] (PDO). Cette librairie implémente une interface entre PHP et n'importe quel système de gestion de base de données (SGBD) tel que Oracle, MySQL, MariaDB ou PostgreSQL. L'utilisation de la librairie PDO présente plusieurs avantages :

- elle permet d'interroger une base de données quel que soit le SGBD. Le fait de changer de SGBD nécessite de modifier une seule ligne de code dans le script PHP,
- elle facilite l'utilisation des requêtes préparées[2] permettant de créer des requêtes plus sécurisées.

1. Il existe d'autres moyens d'accéder à une base de données mais qui deviennent obsolètes par rapport à PDO.
2. Il existe d'autres méthodes pour interroger la base de données qui ne sont pas décrites dans ce cours.

Les requêtes préparées sont des requêtes SQL qui se déroulent en deux temps. La requête est tout d'abord *préparée* par le SGBD, c'est-à-dire analysée et optimisée. La requête est ensuite *exécutée*, c'est-à-dire que la base de données est interrogée et les résultats envoyés au script PHP. Les requêtes préparées peuvent contenir des *marqueurs de place* (paramètres) dont la valeur est donnée au moment de l'exécution de la requête.

Les requêtes préparées permettent d'accélérer l'interrogation de la base de données si une requête préparée est exécutée plusieurs fois (elle ne sera préparée qu'une seule fois). De plus, l'utilisation des marqueurs de place permet une sécurisation de la base de données en diminuant le risque d'attaque par injection SQL.

2.1.1 Connexion à la base de données

L'accès à la base de données se fait à travers l'utilisation d'un objet de la classe `PDO`. La connexion se fait lors de la construction de cet objet. Le constructeur prend en paramètre 3 valeurs (`string`) : le DSN, le login et le mot de passe [3].

Le paramètre *Data Source Name (DSN)* est une chaîne de caractères contenant les informations requises pour se connecter à la base de données : le nom du pilote PDO (nom du SGBD : mysql, pgsql, etc), le nom de la base de données et le nom du serveur sur lequel se trouve cette base. Le DSN suit une syntaxe spécifique :

```
"pilote:host=serveur;dbname=nomBd"
```

À titre d'exemple, si la base de données (SGBD `pgsql`) s'appelle `bdCours` et se trouve sur le serveur local, et si les login et mot de passe sont respectivement `lambda` et `code` , alors la création de l'objet PDO permettant la connexion à la base se fait selon l'instruction :

```
$bd = new PDO('pgsql:host=localhost;dbname=bdCours', 'lambda', 'code');
```

On ajoute juste après l'instanciation de l'objet PDO les instructions

```
$bd->query("SET NAMES 'utf8'");
$bd->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
```

La première instruction indique que le script PHP communique avec la base de données en utf-8. La seconde indique qu'en cas d'erreur, une exception doit être levée, ce qui arrête le script en affichant l'erreur, sauf si l'exception est rattrapée.

! *Il faut se connecter une seule fois à la base de données, même si l'on fait plusieurs requêtes.*

R *Avant de mettre le site Web en production, il faut rattraper les exceptions, notamment celles générées par la librairie PDO, pour ne pas afficher le message d'erreur par défaut. Ceci se fait à l'aide des blocs* `try/catch` *. Pour le développement, le mieux est de ne pas en mettre mais d'utiliser le module* `xdebug` [4].

2.1.2 Interrogation de la base de données

La préparation d'une requête se fait via la méthode `prepare` de la classe `PDO` . Cette méthode prend en paramètre une requête SQL (sous la forme d'une chaîne de caractères) et retourne un objet de la classe `PDOStatement` correspondant à la requête optimisée. En cas d'erreur lors de la préparation, une exception `PDOException` est levée.

La requête est ensuite exécutée grâce à la méthode `execute` de la classe `PDOStatement` . Elle s'applique à l'objet retourné par la méthode `prepare` . En cas d'erreur, une exception est également levée.

3. Les deux derniers paramètres sont optionnels.
4. *c.f.* `https://lucidar.me/fr/aws-cloud9/how-to-install-and-configure-xdebug-on-ubuntu/` pour installer et configurer le module xdebug.

La récupération de la réponse se fait de deux manières : via la méthode `fetch` pour récupérer les résultats un par un, ou via la méthode `fetchAll` pour récupérer tous les résultats en une seule fois.

Méthode `fetch`

La méthode `fetch` retourne la ligne suivante de la réponse, ou **`false`** si toutes les lignes de la réponse ont déjà été retournées. Le type retourné dépend de la valeur passée en paramètre de la méthode `fetch` :

- `PDO::FETCH_ASSOC` : la valeur retournée est un tableau associatif. Les clés sont les noms des colonnes dans la requête et les valeurs celles de la ligne.
- `PDO::FETCH_NUM` : les clés associées aux valeurs sont les indices des colonnes de la requête.
- `PDO::FETCH_BOTH` : le tableau retourné contient deux fois chaque valeur. Les clés sont les noms et les indices des colonnes de la requête (le tableau correspond à l'union des deux premiers tableaux). C'est la valeur par défaut.
- `PDO::FETCH_OBJ` : la valeur retournée est cette fois-ci un objet. Les noms des attributs sont les noms de colonnes de la requête, les valeurs de ces attributs étant les valeurs de la ligne.

Voici un exemple d'interrogation d'une base de données. Supposons que l'on ait la table `Personnes` contenant les lignes suivantes.

Nom	Prenom
Grappe	Roland
David	Julien
Borne	Sylvie
Toulouse	Sophie

L'exécution du code

```php
$requete = $bd->prepare('SELECT Nom, Prenom FROM Personnes');
$requete->execute();
$tab = $requete->fetch(PDO::FETCH_ASSOC);
echo '<p> Nom : ' . $tab['Nom'] . ' et prénom : ' . $tab['Prenom']
↪  .'</p>';
$tab = $requete->fetch(PDO::FETCH_NUM);
echo '<p> Nom : ' . $tab[0] . ' et prénom : ' . $tab[1].'</p>';
$tab = $requete->fetch(PDO::FETCH_BOTH);
echo '<p> Nom : ' . $tab[0] . ' et prénom : ' . $tab['Prenom'].'</p>';
$obj = $requete->fetch(PDO::FETCH_OBJ);
echo '<p> Nom : ' . $obj->Nom . ' et prénom : ' . $obj->Prenom.'</p>';
```

affichera alors

```
Nom : Grappe et prénom : Roland
Nom : David et prénom : Julien
Nom : Borne et prénom : Sylvie
Nom : Toulouse et prénom : Sophie
```

Lors du premier appel, `fetch` retourne le tableau correspondant à la première ligne de la table `Personnes`. Comme la valeur `PDO::FETCH_ASSOC` est passée en argument, le tableau retourné est

```
[
  "Nom"    => "Grappe",
  "Prenom" => "Roland"
]
```

Lors du deuxième appel, `fetch` retourne les informations correspondant à la deuxième ligne de la table car à chaque appel, `fetch` passe automatiquement à la ligne suivante. Comme `PDO::FETCH_NUM` est passé en paramètre, la valeur retournée est

```
[
  0 => "David",
  1 => "Julien"
)
```

Le tableau retourné par le troisième appel est

```
[
  "Nom"     => "Borne",
  0         => "Borne",
  "Prenom"  => "Sylvie",
  1         => "Sylvie"
]
```

Lors du 4ème appel, l'objet retourné contient deux attributs `Nom` et `Prenom` dont les valeurs sont respectivement `Toulouse` et `Sophie`.

Pour chaque ligne de la réponse d'une requête SQL, il faut appeler la méthode `fetch`. Il faut donc utiliser une boucle pour récupérer toutes les lignes. Pour cela, on utilise le fait que la méthode `fetch` retourne **`false`** lorsqu'il n'y a plus de ligne.

L'affichage de toutes les personnes de la table se fait de la manière suivante :

```php
$requete = $bd->prepare('SELECT Nom, Prenom FROM Personnes');
$requete->execute();
while($ligne = $requete->fetch(PDO::FETCH_ASSOC)) {
  echo '<p> Nom : ' . $ligne['Nom'] . ' et prénom : ' . $ligne['Prenom']
       . '</p>';
}
```

R *Le nombre d'appels de `fetch` correspond au nombre de lignes de la réponse + 1 (pour le **`false`**).*

Méthode `fetchAll`

La méthode **`fetchAll`** permet de récupérer toutes les lignes en une seule fois dans un tableau, chaque case de ce tableau correspondant à une ligne de la réponse. Les clés de ce tableau sont les indices ; les valeurs sont soient un tableau, soit un objet selon la valeur passée en paramètre de la méthode **`fetchAll`** (`PDO::FETCH_ASSOC`, `PDO::FETCH_NUM`, `PDO::FETCH_BOTH` ou `PDO::FETCH_OBJ`).

Ainsi, après exécution du code

```php
$requete = $bd->prepare('SELECT Nom, Prenom FROM Personnes');
$requete->execute();
$tab = $requete->fetchAll(PDO::FETCH_ASSOC);
```

la variable **`$tab`** vaut (si la table **`Personnes`** n'a pas changé) :

```
$tab = [
  0 => ['Nom' => 'Grappe',   'Prenom' => 'Roland'],
  1 => ['Nom' => 'David',    'Prenom' => 'Julien'],
  2 => ['Nom' => 'Borne',    'Prenom' => 'Sylvie'],
  3 => ['Nom' => 'Toulouse', 'Prenom' => 'Sophie']
]
```

La réponse est donc un tableau de tableaux.

R *Il faut faire attention avec l'utilisation de la méthode `fetchAll` : comme tous les résultats sont retournés en même temps, l'espace mémoire occupée par `$tab` peut être potentiellement important suivant le nombre de lignes de la réponse. Pour des très grandes bases de données, on préférera utiliser la méthode `fetch` ou inclure une limite (clause `LIMIT`) dans la requête.*

Connaître le nombre de lignes modifiées

Certaines requêtes SQL (**DELETE**, **INSERT** ou **UPDATE**) peuvent modifier la base de données. Pour connaître le nombre de lignes affectées par la dernière modification (dernier appel de la méthode `execute`), on utilise la méthode `rowCount` de la classe `PDOStatement`.

```php
$requete = $bd->prepare('DELETE FROM Personnes');
$requete->execute();
$nb = $requete->rowCount();
echo "<p> $nb personnes ont été supprimées </p>";
```

R *Cela ne fonctionne pas avec le pilote SQLite et ne fonctionne avec PostgreSQL uniquement quand l'attribut de déclaration `PDO::ATTR_CURSOR` est défini à `PDO::CURSOR_SCROLL`.*

2.1.3 Marqueurs de place

Les marqueurs de place permettent de marquer des endroits dans la requête SQL qui seront remplacés par des valeurs au moment de l'exécution de cette requête. Les marqueurs permettent donc d'exécuter plusieurs fois une même requête avec des valeurs de marqueurs différentes. De plus, lorsque les marqueurs sont utilisés pour insérer dans la requête SQL des valeurs saisies par l'utilisateur, ils permettent de sécuriser la base de données des attaques par injection SQL.

Dans une requête SQL, un marqueur de place a un nom précédé du symbole ":". Avant l'exécution de la requête, il faudra alors donner une valeur à tous les marqueurs de place contenus dans la requête, grâce à la méthode `bindValue`. Cette méthode prend en paramètre un nom de marqueur et sa valeur [5].

Voici un exemple de requête préparée avec des marqueurs de place [6] :

```php
$couleur = 'rouge';
$req = $bd->prepare('SELECT nom, couleur, calories
                     FROM fruit
                     WHERE calories < :calories AND couleur = :couleur');
$req->bindValue(':calories', 150);
$req->bindValue(':couleur', $couleur);
$req->execute();
```

Lors de l'exécution, les marqueurs `':calories'` et `':couleur'` sont respectivement remplacés par les valeurs `150` et `rouge`. La requête est donc :
`SELECT nom, couleur, calories FROM fruit WHERE calories < 150 AND couleur='rouge'`

2.1.4 Organisation du code : bonne pratique

Pour simplifier le développement et la maintenance d'un site, il est important de regrouper au même endroit tous les accès à la base de données et de séparer les traitements des données de l'affichage de la page Web.

5. La méthode `bindValue` peut également prendre un troisième paramètre indiquant le type de la valeur (`PDO::PARAM_INT`, `PDO::PARAM_STR`, etc). Ce paramètre est parfois nécessaire pour construire des requêtes syntaxiquement correctes (notamment avec la clause `LIMIT`).
6. Exemple issu du site PHP.net

Dans ce cours, on utilisera une classe appelée `Model`[7] pour accéder à la base de données. Le constructeur de `Model` créera un objet `PDO` et le stockera dans un attribut privé `$bd`. Chaque requête correspondra à une méthode de classe. Celles-ci utiliseront l'attribut `$bd` pour accéder à la base de données. Les méthodes récupérant des données dans la base de données les retourneront sous forme de tableaux PHP. Cette classe ne contiendra aucun code HTML.

Chaque objet PDO créé correspond à une connexion avec le SGBD. Or, avoir plusieurs connexions en parallèle pour un même script surcharge inutilement le serveur et réduit ses performances. Il faut donc s'assurer que chaque script PHP se connecte au plus une fois à la base de données. Pour cela, la classe `Model` implémentera le design pattern singleton. Celui-ci assure qu'un seul objet de la classe `Model` pourra être instancié dans un script PHP.

Pour implémenter le design pattern singleton, on rend le constructeur de la classe privé et on ajoute à cette classe un attribut statique privé `$instance`. Cet attribut statique initialisé à `null` contiendra par la suite l'unique instanciation de la classe `Model`. On ajoute une méthode statique publique `getModel()` qui retourne l'instance de `Model` stockée dans `$instance`. Lors du premier appel, `getModel()` appelle le constructeur de la classe `Model` pour initialiser `$instance`.

Voici un squelette de la classe `Model`.

```php
class Model{

  private $bd; //Attribut privé contenant l'objet PDO

  //Attribut qui contiendra l'unique instance du modèle
  private static $instance = null;

  /**
  * Constructeur créant l'objet PDO et l'affectant à $bd
  */
  private function __construct()
  {
      require "/home/Web/Auth/credentials.php";
      $this->bd = new PDO($dsn, $login, $mdp);
      $this->bd->query("SET NAMES 'utf8'");
      $this->bd->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
  }

  /**
  * Méthode permettant de récupérer l'instance de la classe Model
  */
  public static function getModel()
  {
    //Si la classe n'a pas encore été instanciée
    if (self::$instance === null) {
        self::$instance = new self(); //On l'instancie
    }
    return self::$instance; //On retourne l'instance
  }
}
```

Pour accéder à la base de données, on récupère l'objet de la classe modèle de la manière suivante.

```php
$m = Model::getModel(); //On récupère le modèle
//On appelle une méthode d'accès à la base de données (à définir)
$data = $m->getData();
```

7. Ce nom vient de l'architecture MVC que l'on verra plus tard.

Cette organisation du code convient surtout dans le cas de petites bases de données (peu de tables). Pour les grandes bases de données, il faudrait sûrement créer plusieurs classes. Il est également possible d'utiliser un ORM *(Object Relation Mapping) qui est un ensemble de librairies qui permettent d'interagir avec les bases de données via des objets. Différents ORM existent : EZPDO, Doctrine, Propel, etc. L'ORM permet de simplifier grandement l'accès aux données : chaque tuple devient une instance d'objet et les méthodes de modification sur les données deviennent uniformisées. Voici un exemple de requête via un ORM (ici Doctrine) :*

```
//Récupérer tous les articles dont le titre est 'Mon titre'
$articles = $table->findByTitle("Mon titre");
```

Il faut noter cependant que la configuration d'un ORM peut être complexe et que les ORM ont un coût en termes de performance.

2.2 Sécurité

Il existe de nombreuses failles de sécurité qui peuvent être exploitées par des pirates pour accéder à des données confidentielles, exécuter du code sur le serveur ou d'autres navigateurs, etc. Il faut donc faire très attention lors du développement d'un site Web aux aspects sécurité. Cette partie présente plusieurs failles de sécurité couramment exploitées pour pirater des sites Web ou les bases de données de ces sites, ainsi que la protection contre ces failles.

Il existe bien d'autres problèmes de sécurité non abordés dans ce cours auxquels il faut faire attention : frameworks obsolètes, failles CSRF, vol de session, etc. L'OWASP (Open Web Application Security Project) publie le TOP 10 des failles de sécurité[8] *qu'il faut prendre en compte (au minimum) lors du développement et maintien d'un site Web.*

2.2.1 Faille SQLi

La faille SQLi (pour faille injection SQL) est une faille de sécurité qui peut apparaître lors de l'interaction avec une base de données. Elle permet une attaque par injection SQL qui consiste à modifier une requête SQL pour obtenir un résultat différent de celui prévu. Supposons que l'on ait une table `Commandes` dans la base de données contenant les commandes de chaque client. Cette table pourrait être du type[9] :

Id	IdC	NomP	Qte
1	1	Canapé	1
2	1	Lit	2
3	2	Table	1
4	2	Voiture	1

Le champ `Id` correspond à la clé primaire, `IdC` est l'identifiant du client (clé étrangère), et `NomP` et `Qte` désignent respectivement le nom du produit et la quantité demandée.

Pour afficher la liste des commandes d'un utilisateur, on pourrait être tenté d'écrire le code PHP suivant :

```
1 if(isset($_GET['id'])) {
2     $req = $bd->prepare('SELECT * FROM Commandes WHERE IdC = ' .
      ↪  $_GET['id']);
3     $req->execute();
4     while($t = $req->fetch(PDO::FETCH_ASSOC))
5       echo '<p>' . $t['nomP'] . ' ' . $t['Qte'] . '</p>';
6 }
```

8. `https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project`
9. Cette table est simpliste mais suffisante pour montrer la faille de sécurité.

Ainsi, l'appel du script via l'url `http://serveur/script.php?id=1` afficherait les deux premières lignes de la table puisque l'identifiant du client est 1. Cependant, si le script est appelé via l'url `http://serveur/script.php?id=1;DELETE * FROM Commandes`, la "requête" devient :

```
SELECT * FROM Commandes WHERE IdC = 1;DELETE * FROM Commandes
```

Ce code SQL est interprété comme deux requêtes : afficher toutes les commandes du client 1 puis supprimer toutes les valeurs de la table !

Il existe plusieurs méthodes pour se protéger de telles attaques. On peut protéger la valeur donnée par l'utilisateur dans une requête SQL en la mettant entre apostrophes, grâce à la méthode `quote`. Ainsi, en remplaçant la ligne 3 par :

```
$req = $bd->prepare('SELECT * FROM Commandes WHERE IdC = '
       . $bd->quote($_GET['id']));
```

la requête avec l'url précédente devient :

```
SELECT * FROM Commandes WHERE IdC = '1;DELETE * FROM Commandes'
```

Il n'y a donc qu'une requête SQL qui affiche les commandes appartenant à l'utilisateur dont l'identifiant est `'1;DELETE * FROM Commandes'`.

Une façon beaucoup plus efficace (et fortement conseillée) est d'utiliser un marqueur de place.

```
$req = $bd->prepare('SELECT * FROM Commandes WHERE IdC = :ident');
$req->bindValue(':ident',$_GET['id']);
```

Ainsi, la valeur saisie par l'utilisateur est automatiquement protégée car elle est comprise comme une simple valeur et non comme une valeur contenant du code SQL.

2.2.2 Faille XSS

Une autre faille de sécurité importante est la faille XSS (pour *cross-site scripting*). Elle permet à un utilisateur malveillant d'insérer du code javascript [10] (ou autre) qui sera interprété par le navigateur. Supposons que l'on ait une table `Personnes` identique à celle donnée par la table 2.1.

Nom	**Prenom**
Grappe	Roland
David	Julien
Malveillant	<script>confirm("attaque");</script>

TABLE 2.1 – Table `Personnes`

Il est clair que la dernière ligne de cette table n'est pas correcte. Un utilisateur malveillant a inséré du code javascript pour la valeur du champ `Prenom`. Supposons maintenant qu'un script PHP affiche toutes les personnes de la table (c.f. code de la section 2.1.2). Lorsque le script sera exécuté, il enverra au navigateur le code HTML

```
<p> Nom : Grappe et prénom : Roland </p>
<p> Nom : David et prénom : Julien </p>
<p> Nom : Malveillant et prénom : <script>confirm("attaque");</script> </p>
```

Le navigateur recevant ce code HTML interprétera la balise `script` (balise HTML indiquant que ce qui est à l'intérieur est un code javascript à exécuter) et exécutera le code javascript :

10. Le langage javascript est un langage *client*, c'est-à-dire s'exécutant sur le navigateur.

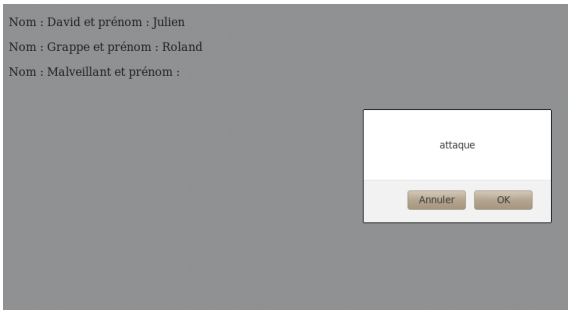

FIGURE 2.2 – Interprétation du code HTML reçu (faille XSS)

Autrement dit, toutes les personnes qui demanderont cette page PHP auront déclenché, à leur insu et à celle du développeur du site, l'exécution d'un code javascript. Bien évidemment, ce code peut être bien plus gênant que l'affichage d'une fenêtre ! Ce code peut par exemple [11] :

- faire une redirection (parfois de manière transparente) de l'utilisateur (souvent dans un but de hameçonnage),
- voler des informations, par exemple les sessions et les cookies,
- effectuer des actions sur le site faillible, à l'insu de la victime et sous son identité (envoi de messages, suppression de données...),
- rendre la lecture d'une page difficile (boucle infinie d'alertes par exemple).

Pour se prémunir d'une telle attaque, il faut désactiver dans le code HTML généré toutes les balises HTML qui ne sont pas prévues. Pour cela, on utilise

```
htmlspecialchars($ch, ENT_QUOTES)
```

qui retourne une chaîne de caractères identique à `$ch`, sauf que certains caractères spéciaux HTML sont remplacés par leur code HTML. Par exemple, le caractère `<` sera remplacé par son code `<`. L'appel de la fonction

```
htmlspecialchars('<script>confirm("attaque");</script>', ENT_QUOTES)
```

retournera `<script>confirm(&quot;attaque&quot;);</script>`.

Dans le code PHP affichant la liste des personnes, remplacer l'instruction :

```
echo '<p> Nom : ' . $rep['Nom'] . ' et prénom : ' . $rep['Prenom'] .
 ↪  '</p>';
```

par

```
echo '<p> Nom : ' . htmlspecialchars($rep['Nom'], ENT_QUOTES) .
    ' et prénom : ' . htmlspecialchars($rep['Prenom'], ENT_QUOTES) .
     ↪  '</p>';
```

générera le code

```
<p> Nom : Grappe et prénom : Roland </p>
<p> Nom : David et prénom : Julien </p>
<p> Nom : Malveillant et prénom : &lt;script&gt;confirm(&quot;attaque&quot;);
&lt;script&gt; </p>
```

Le navigateur n'exécutera aucun code javascript et, interprétant les caractères spéciaux, affichera

11. Exemples donnés sur Wikipédia.

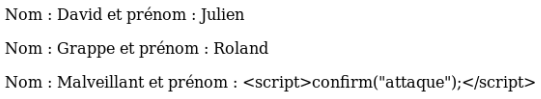

FIGURE 2.3 – Interprétation du code HTML reçu (avec correction de la faille XSS)

Il faut TOUJOURS s'assurer que les données saisies par l'utilisateur soient affichées avec la fonction `htmlspecialchars` *lors de l'instruction* `echo`.

Comme le caractère & est lui-même un caractère HTML spécial (code `&` *), il ne faut appliquer* `htmlspecialchars` *qu'une seule fois.*

2.2.3 Gestion des mots de passe

Il est souvent nécessaire, dans un site Web, que les utilisateurs s'authentifient en donnant un login et un mot de passe. Dans ce cas, la base de données doit contenir ces informations. Lors de l'authentification, si le login saisi par l'utilisateur existe dans la base de données, le script PHP vérifie que le mot de passe saisi par l'utilisateur est bien le même que celui associé au login dans la base de données. On serait donc tenté d'avoir une table `MotsDePasse` ressemblant à :

login	pass
jim	azerty
arn	php123

Ainsi, le script d'authentification serait :

```php
//On suppose que les login et mot de passe saisis par l'utilisateur sont
//dans $_POST['login'] et $_POST['pass']
$req = $bd->prepare('SELECT pass FROM MotsDePasse WHERE login=:login');
$req->bindValue(':login',$_POST['login']);
$req->execute();
$res = $req->fetch(PDO::FETCH_ASSOC);
//S'il y a une correspondance
if($res) {
  if($_POST['pass']==$res['pass']) { //Si les mots de passe sont les mêmes
    echo '<p> Connexion réussie </p>';
    $_SESSION['connecte'] = true;
  }
}
```

Si la portion de script PHP ne présente pas de faille, stocker les mots de passe en clair dans la base de données n'est pas sécurisé. En effet, si un individu malveillant accède à la table `MotsDePasse` (grâce à une faille dans le site Web [12] ou parce qu'il a les droits pour accéder à cette table), il peut se connecter avec n'importe quel compte. De plus, comme les gens utilisent généralement les mêmes mots de passe pour différents sites Web, il peut usurper le compte d'un des utilisateurs sur un autre site. **Il ne faut donc pas stocker les mots de passe en clair dans une base de données.**

Pour cela, on chiffre (ou crypte) les mots de passe avant de les stocker. En PHP, on utilise la fonction `password_hash`. Cette fonction prend en paramètre une chaîne de caractères et un algorithme de chiffrement. On choisira l'algorithme de chiffrement par défaut (constante `PASSWORD_DEFAULT`)

12. Il existe par exemple des logiciels tels que SQLi Dumper qui permettent de récupérer automatiquement en quelques minutes des dizaines/centaines de bases de données de sites Web ayant une faille SQLi.

correspondant à l'algorithme Bcrypt. La fonction renvoie la chaîne correspondant au hachage (chiffrement) de la chaîne passée en paramètre selon l'algorithme spécifié. À titre d'exemple, l'appel `password_hash("azerty", PASSWORD_DEFAULT)` renvoie :

`'$2y$10$PKXDDhEOPN.OlZUHqEKX5eH5no.ci1ssfhIW5.PfU40VeMr4ftBp2'`.

L'intérêt des fonctions de hachage cryptographique (ou fonctions de chiffrement) est qu'il est très facile de chiffrer une chaîne mais très difficile de la décrypter, c'est-à-dire de retrouver `'azerty'` à partir de `'$2y$10$PKXDDhEOPN.OlZUHqEKX5eH5no.ci1ssfhIW5.PfU40VeMr4ftBp2'`. Les mots de passe dans la base de données doivent donc être stockés déjà chiffrés. On obtient alors la table :

login	**pass**
jim	$2y$10$PKXDDhEOPN.OlZUHqEKX5eH5no.ci1ssfhIW5.PfU40VeMr4ftBp2
arn	$2y$10$4ntvCh7GUrAUbAUESzddHu9g/9/PyATF7NjUvZhRvCbVEN03GW9Hm

FIGURE 2.4 – Table SQL contenant les mots de passe chiffrés.

L'individu accédant à la table ne pourra donc pas connaître les mots de passe, il n'aura accès qu'aux mots de passe chiffrés. Cependant, il pourra générer un très grand nombre de mots de passe chiffrés et comparer les résultats obtenus avec les mots de passe de la table (attaque par force brute [13]). En cas de correspondance, il aura donc trouvé un mot de passe.

R *La fonction `password_hash` utilise un grain de sel pour générer un mot de passe. Le grain de sel est une chaîne de caractères générée aléatoirement qui est utilisée lors du chiffrement du mot de passe. Elle est stockée dans le mot de passe crypté [14]. Ceci permet que deux mots de passe identiques n'aient pas le même chiffrement. Ceci évite les attaques par dictionnaire qui consistent à comparer les mots de passe cryptés avec un ensemble de chaînes correspondant au chiffrement des mots de passe les plus courants.*

Pour tester si un mot de passe de correct, il faut appeler la fonction `password_verify` qui prend en paramètre le mot de passe et le mot de passe crypté et retourne `true` si les mots de passe correspondent et `false` sinon.

Dans le script d'authentification, le test de la ligne 9 sera remplacé par :

```
if(password_verify($_POST['pass'],$res['pass']))
```

13. Une fonction de chiffrement "intéressante" est une fonction dont le temps d'exécution (pour crypter un mot de passe) est suffisamment rapide pour l'utiliser une fois dans un script PHP sans le ralentir mais trop long pour crypter un grand nombre de mots de passe dans le cas d'une attaque par force brute.

14. Dans les mots de passe cryptés avec `password_hash`, les premiers caractères indiquent l'algorithme utilisé ainsi que le nombre d'itérations de chiffrement. Les caractères suivants correspondent au grain de sel et le reste de la chaîne correspond au chiffrement du mot de passe en fonction de l'algorithme et du grain de sel.