

Gestion des utilisateurs

Table des matières

I. Contexte	3
II. Les données utilisateur	3
III. Exercice : Appliquez la notion	5
IV. L'ID	6
V. Exercice : Appliquez la notion	8
VI. Les identifiants	8
VII. Exercice : Appliquez la notion	11
VIII. Les rôles	12
IX. Exercice : Appliquez la notion	15
X. Essentiel	17
XI. Auto-évaluation	17
A. Exercice final	17
B. Exercice : Défi.....	19
Solutions des exercices	20

I. Contexte

Durée : 1 h

Environnement de travail : Local

Pré-requis : Savoir utiliser PDO et connaître la POO

Contexte

Que ce soit Facebook, Leboncoin, Google ou même votre plateforme de formation en ligne préférée, tous ces sites ont un point commun : il est possible de s'y créer un compte utilisateur. Gérer des utilisateurs est un aspect de la programmation que l'on retrouve dans la grande majorité des projets informatiques. Il est donc intéressant d'approfondir le sujet pour en connaître les bases.

Dans ce cours, nous allons voir quelles sont les principales informations nécessaires à la gestion des utilisateurs en base de données, mais aussi comment gérer les actions communes, comme la connexion ou la gestion des droits.

II. Les données utilisateur

Objectifs

- Savoir quelles données sont nécessaires à la gestion des utilisateurs
- Différencier les données techniques des données métier

Mise en situation

Il existe deux types de données rattachées à notre utilisateur : les **données techniques**, qui sont toutes les données dont notre application va avoir besoin afin de gérer les utilisateurs, et les **données métier**, qui sont les informations plus spécifiques au contexte du projet. Cette partie va nous permettre de différencier les deux et de poser les bases du reste du cours.

Définition Les données techniques

Les données techniques sont les données qui vont nous permettre de gérer les utilisateurs et de réaliser toutes les actions que l'on souhaite faire dessus, comme la connexion ou la gestion des droits.

Méthode

Il existe 4 principales données techniques qui répondent à la majorité des besoins :

- L'**ID**, c'est-à-dire l'identifiant technique unique qui va permettre de distinguer un utilisateur. Il est généré par l'application et n'est jamais manipulé directement par l'utilisateur. Il n'est pas modifiable et représente donc la clé primaire de la table de nos utilisateurs.
- Le **login**, qui est un identifiant unique fourni par l'utilisateur. Il est utilisé pour la connexion et vient pallier le fait que l'identifiant technique, bien que très facile à manipuler pour les machines, est complexe à retenir pour un être humain. L'utilisateur final va donc renseigner son login et c'est à la charge de notre application de récupérer l'ID associé.
- Le **mot de passe**, qui permet de sécuriser un compte utilisateur. Il est généré par l'application, mais peut être modifiable par l'utilisateur. Il est utilisé conjointement avec le login au moment de la connexion et permet de s'assurer que l'utilisateur qui essaye de se connecter n'est pas un usurpateur.

- Enfin, le **rôle** permet de définir les différents droits de notre utilisateur, c'est-à-dire à quelle page ou à quelle fonctionnalité il aura accès. Par exemple, un administrateur n'aura pas les mêmes accès qu'un utilisateur normal. Les rôles de base sont assignés par l'application, mais, bien souvent, les rôles plus avancés sont attribués par les autres utilisateurs. Ainsi, seuls les administrateurs pourront décider d'attribuer le rôle d'administrateur à un autre utilisateur.

Ces quatre éléments représentent la base de la gestion des utilisateurs. Ils sont présents dans la majorité des cas, quel que soit le contexte du projet, même s'il est possible de trouver des variantes dans de très rares cas (un petit site sans gestion des droits, un login composé pour permettre les doublons ou une interface administrateur sans login, par exemple).

Définition Les données métier

Les données métier sont toutes les autres données rattachées à l'utilisateur qui vont changer d'une application à une autre. Ces données peuvent être diverses et variées : on peut avoir besoin du nom et du prénom de l'utilisateur, de son adresse, le numéro de sa carte bleue, sa langue, etc.

Contrairement aux données techniques, il n'y a pas de règle particulière pour gérer les données métier : dans chaque cas, la forme et le type de donnée est différent. Il est donc important de déterminer son format et de s'assurer qu'il est toujours respecté, en mettant en place des vérifications. Cela peut être simple en apparence, mais certaines données possèdent une logique métier qui peut être plus complexe qu'il n'y paraît. Il faut également savoir si la donnée est obligatoire ou non et prévoir la structure de sa base de données et ses vérifications en conséquence.

Exemple

Si on doit stocker un numéro de carte bleue, il faut s'assurer que ce numéro soit bien valide. Pour cela, une vérification simple est de s'assurer que le numéro est bien de la forme XXXX-XXXX-XXXX-XXXX, chaque X représentant une valeur numérique.

Cependant, cette vérification est très simpliste et laisserait passer des numéros qui ne correspondent pas à de vrais numéros de carte bleue. En effet, la règle métier est en réalité plus complexe :

- Les 6 premiers nombres correspondent à l'identifiant de la banque
- Les 9 nombres qui suivent représentent le numéro de la carte
- Le dernier chiffre est un nombre de vérification, déterminé en appliquant l'algorithme de Luhn sur le reste des nombres

L'algorithme de Luhn est un algorithme de vérification qui consiste à prendre une série de nombres, doubler un nombre sur deux en commençant par le second et calculer le modulo de cette somme. Il permet de repérer les erreurs accidentelles au moment de la saisie d'un numéro de carte.

Pour que notre vérification soit efficace, il nous faut donc implémenter cet algorithme pour vérifier le numéro saisi par l'utilisateur. Si c'est possible, on pourrait même vérifier que les 6 premiers nombres correspondent à un véritable établissement bancaire.

Enfin, même si ce sont des cas rares, un numéro de carte bleue peut en réalité posséder entre 8 et 19 nombres, ce qui change complètement la structure de base.

Syntaxe **À retenir**

- Les données techniques sont les données nécessaires au fonctionnement de la gestion des utilisateurs. Elles sont au nombre de 4 : l'ID, le login, le mot de passe et le rôle.
- Les données métier sont les autres données rattachées aux utilisateurs et qui dépendent de l'application, comme le nom, le numéro de carte bleue, etc. Il faut toujours faire attention à l'implémentation des règles métier, qui peuvent parfois être plus complexes qu'il n'y paraît.

ComplémentFormat d'un numéro de carte bleue¹Algorithme de Luhn²

III. Exercice : Appliquez la notion

Question

[solution n°1 p.21]

Vous êtes chargé d'implémenter la saisie d'un numéro de sécurité sociale sur une application. Voici la composition d'un numéro de sécurité sociale, et donc les vérifications à effectuer :

- Le premier chiffre désigne le sexe d'une personne : il ne peut être qu'égal à 1 ou 2
- Les deux chiffres qui suivent représentent l'année de naissance, qui peut être comprise entre 00 et 99
- Les deux chiffres qui suivent représentent le mois de naissance, donc compris entre 01 et 12
- Les 5 chiffres qui suivent diffèrent selon si la personne est née en France, en Outre-mer ou à l'étranger (pour simplifier cet exercice, nous allons simplement vérifier qu'il n'y a que des chiffres)
- Les trois chiffres suivants sont le numéro de naissance, entre 000 et 999
- Enfin, les deux derniers chiffres représentent une clé de contrôle permettant de repérer les fautes de frappe. Ils sont calculés grâce à l'opération : $97 - (13 \text{ premiers chiffres modulo } 97)$

Codez une fonction PHP permettant de vérifier si un numéro de sécurité sociale est valide ou non. Voici quelques cas de test pour vous guider :

- 369054958815780 n'est pas un numéro valide : il commence par 3, qui n'est pas une valeur autorisée
- 269134958815780 n'est pas un numéro valide : le mois de naissance est 13
- 26914958815780 n'est pas un numéro valide : il n'est pas assez long (l'erreur ici est d'avoir mis le mois 1 au lieu de 01)
- 269054958815781 n'est pas un numéro valide : la clé de contrôle ne correspond pas
- 269054958815780 est un numéro valide

Remarque : dans la pratique, les numéros de sécurité sociale sont stockés sous forme de chaîne de caractère. Votre fonction devra donc prendre en paramètre un string.

Indice :

Pour la structure générale, vous pouvez utiliser une expression régulière. Cependant, la clé de contrôle devra être calculée à la main.

Indice :

La fonction `substr` permet de récupérer très facilement des sous-chaînes de caractères.

1 https://en.wikipedia.org/wiki/Payment_card_number

2 https://fr.wikipedia.org/wiki/Formule_de_Luhn

IV. L'ID

Objectifs

- Connaître les différentes manières de gérer un identifiant technique
- Savoir charger un utilisateur à partir de son ID

Mise en situation

L'ID est l'**identifiant unique** qui va nous permettre de distinguer un utilisateur dans notre table : c'est sa clé primaire. Ce champ va nous permettre de charger toutes les informations de notre utilisateur : dans nos requêtes, nous n'allons pas demander les informations de l'utilisateur **John**, mais plutôt de l'utilisateur numéro 36, à quelques exceptions près.

Au moment de créer un identifiant pour un utilisateur, nous avons deux possibilités : utiliser un entier auto-incrémenté, ou un UUID.

Les entiers auto-incrémentés

L'entier auto-incrémenté est ce que nous avons utilisé jusqu'à présent comme clé primaire : un nombre initialisé à 1 et dont la valeur est calculée automatiquement par le SGBD. L'avantage d'utiliser un tel identifiant est qu'il est très simple à mettre en place : tous les moteurs de base de données supportent cette fonctionnalité.

```
1 CREATE TABLE users
2 (
3     id INTEGER(11) NOT NULL PRIMARY KEY AUTO_INCREMENT
4 );
5
6 INSERT INTO users VALUES (NULL);
```

Cependant, cette solution possède quelques inconvénients :

- Tout d'abord, sur de très larges projets nécessitant des systèmes distribués (c'est-à-dire plusieurs serveurs disposant chacun de leur propre base de données), l'auto-incrémentation peut amener à des conflits. En effet, si deux utilisateurs créent un compte au même moment sur deux serveurs différents, ils auront le même ID, ce qui peut poser problème au moment de la fusion des données.
- Ensuite, l'auto-incrémentation est prédictible : si l'URL du profil d'un utilisateur est `/profile/36`, alors on se doute que les utilisateurs 37 ou 35 existent et il est facile d'accéder à leurs profils. Si une bonne gestion des droits permettrait de limiter les risques, cela n'empêcherait pas un utilisateur mal intentionné de créer un script parcourant tous les profils afin de récolter leurs informations publiques.
- Enfin, l'auto-incrémentation divulgue des informations sur le contenu des tables : si quelqu'un souhaite savoir combien d'utilisateurs sont sur l'application, il leur suffit de créer un utilisateur et de regarder l'identifiant généré. Cette information peut être sensible dans certaines situations et représenter un contre-argument commercial.

Pour répondre à ces problématiques, il existe une alternative : les UUID.

Définition **UUID**

Les UUID, pour *Universally Unique IDentifiers*, sont des chaînes de caractères uniques de 36 caractères, composées de cinq nombres au format hexadécimal séparés par des tirets. La taille de chaque nombre suit le standard 8-4-4-4-12 : le format d'un UUID est donc `aaaaaaaa-bbbb-cccc-dddd-eeeeeeeeeeeeee`. Chaque UUID est généré aléatoirement au moment de l'insertion.

Le support de l'UUID diffère selon les SGBD. Certains moteurs de base de données disposent de fonctions permettant de s'occuper de la génération, mais ne gèrent pas le type UUID en tant que tel. C'est le cas, par exemple, de MySQL et de MariaDB : ils disposent d'une fonction `UUID()` permettant de générer un identifiant, mais le champ de la table sera typé comme une chaîne de caractères.

```
1 CREATE TABLE users
2 (
3     id CHAR(36) NOT NULL PRIMARY KEY
4 );
5
6 INSERT INTO users VALUES (UUID());
```

D'autres SGBD, en revanche, supportent le type UUID, mais laissent la génération à l'application : c'est le cas de PostgreSQL.

```
1 CREATE TABLE users(
2     id UUID NOT NULL PRIMARY KEY
3 );
4
5 INSERT INTO users VALUES ('b718bf4d-cb64-11ea-bbef-1831bf21e339'); -- Valeur calculée par
    l'application appelante
```

Remarque Se fier à l'aléatoire

À cause de leur caractère aléatoire, il serait logique de penser que les UUID ne sont pas des clés primaires fiables. En effet, la génération étant aléatoire, il y a un risque de collision, c'est-à-dire de générer une clé qui existerait déjà en base.

Dans la pratique, la complexité des UUID fait qu'il est presque impossible de se retrouver dans cette situation : il existe 2^{112} UUID possibles. Ainsi, la probabilité de générer une clé dupliquée dans une table contenant 103 billions d'UUID (donc 10^{12} enregistrements) est d'une sur un milliard (soit 10^9).

En comparaison, la probabilité de gagner au loto est d'un peu moins de 20 millions, soit 10^6 .

Les UUID permettent de régler les problèmes des entiers auto-incrémentés, au détriment de la lisibilité des URL et d'une mise en place souvent plus complexe. Le choix de l'un ou de l'autre devra donc se faire selon les besoins et les outils disponibles.

Dans le cadre de ce cours, nous allons utiliser MariaDB, qui supporte la génération d'UUID nativement. Nous allons donc utiliser les UUID pour identifier nos utilisateurs.

Fondamental

Le choix entre auto-incrémentation et UUID n'est pas exclusif aux utilisateurs : cette réflexion concerne toutes les clés primaires de la base de données. Parfois, la prédictibilité des auto-incrémentations est une force : pour des articles de blogs, par exemple, elle peut permettre aux utilisateurs de modifier l'URL pour se rendre rapidement à l'article de leur choix. En revanche, pour identifier des commandes sur un site de vente en ligne, cela pourrait permettre à un concurrent de déterminer le volume des ventes en suivant l'évolution des identifiants.

Syntaxe À retenir

- Il existe deux façons de générer des identifiants : les entiers auto-générés et les UUID. Chacun a ses forces et ses faiblesses : les entiers sont faciles à manipuler et à mettre en place, mais prédictibles et non-unique sur différentes bases, tandis que les UUID sont plus complexes, mais aléatoires et complètement uniques.

Complément

UUID¹

Why auto-increment is a terrible idea : un article sur les avantages des UUID²

V. Exercice : Appliquez la notion

Question 1

[solution n°2 p.21]

Une entreprise de transports en commun spécialisée dans les trajets en bus souhaiterait informatiser son système. Pour cela, ils ont besoin d'une base de données MariaDB permettant de gérer leurs **véhicules** et leurs **conducteurs**.

- Un véhicule est composé d'une **plaque d'immatriculation** (sur 9 caractères) et d'une **marque**.
- Un conducteur possède un **nom** et un **prénom**.
- Le manager organise les **voyages** : un **conducteur** est assigné à un **véhicule** pour une certaine **date**.

L'entreprise possède plusieurs bureaux dans des villes différentes. Pour le moment, chaque bureau va posséder sa propre instance de l'application, sur des bases de données séparées. Cependant, le but de l'entreprise, à long terme, est de fusionner toutes les données dans une seule base, mais cela représente un coût trop élevé pour le moment.

À partir de ces informations, créez un schéma de base de données correspondant aux attentes de l'entreprise. Vous êtes libre de rajouter autant de champs que vous le souhaitez.

Indice :

La plaque d'immatriculation n'est pas une bonne clé primaire : elle peut changer.

Indice :

Le fait que l'entreprise veuille fusionner les données en une seule base est une forte indication des bénéfices de l'UUID sur l'auto-incrémentation.

Question 2

[solution n°3 p.22]

Rédigez les requêtes SQL permettant d'insérer un véhicule et un conducteur de votre choix.

Indice :

La fonction `UUID()` permet de générer des UUID sur MariaDB.

VI. Les identifiants

Objectifs

- Stocker les identifiants d'un utilisateur
- Connecter un utilisateur à l'application

Mise en situation

Pour qu'un utilisateur puisse se connecter, nous n'allons pas lui demander son identifiant technique : c'est non seulement une valeur trop complexe à retenir pour un humain, mais aussi une donnée qui n'est pas sécurisée. À la place, nous allons lui demander un **identifiant** et un **mot de passe**, qu'il aura préalablement choisis au moment de l'inscription. Cela nous permettra de nous assurer que l'utilisateur qui essaie de se connecter est bien celui qu'il prétend être.

¹ [https://en.wikipedia.org/wiki/Universally_unique_identifier#Version_4_\(random\)](https://en.wikipedia.org/wiki/Universally_unique_identifier#Version_4_(random))

² <https://www.clever-cloud.com/blog/engineering/2015/05/20/why-auto-increment-is-a-terrible-idea/>

Méthode **Le login**

Le **login** (ou « nom d'utilisateur ») est un nom choisi par l'utilisateur qui représente son identité sur notre application. Il doit être unique pour éviter les risques d'usurpation d'identité.

Cependant, un login n'est pas un bon candidat pour une clé primaire. En effet, les noms d'utilisateurs sont amenés à changer : même si ce n'est pas une option proposée aux utilisateurs, il faut toujours prévoir les cas où un nom d'utilisateur serait offensant et qu'un administrateur doive le modifier.

Un login est représenté en base de données par une chaîne de caractères de taille variable, avec une limite entre 20 et 50 caractères. Cette limite est importante, car le nom d'utilisateur est une donnée qui va être affichée sur le site : il faut donc prévoir un affichage en accord avec la taille maximale de cette donnée.

```
1 CREATE TABLE users
2 (
3     login VARCHAR(20) NOT NULL UNIQUE
4 );
```

L'unicité du login est parfois frustrante pour les utilisateurs, qui ne peuvent pas utiliser leur pseudonyme habituel. Pour pallier ce problème, une adresse e-mail est un très bon candidat pour un login. En effet, c'est une donnée qui est déjà unique par nature et c'est une donnée dont beaucoup d'applications ont de toute façon besoin : elle est très souvent utilisée pour confirmer l'inscription de l'utilisateur via un lien d'activation, par exemple. La taille maximale d'un e-mail est de 254 caractères.

En utilisant une adresse e-mail comme login, il est alors possible de laisser l'utilisateur choisir un nom qui sera affiché dans l'application, mais qui n'aura aucun lien avec la connexion. Ce nom d'affichage pourra ne pas être unique, mais conservera sa contrainte de taille.

```
1 CREATE TABLE users
2 (
3     email VARCHAR(254) NOT NULL UNIQUE,
4     username VARCHAR(20) NOT NULL
5 );
```

Méthode **Le mot de passe**

Le mot de passe est une chaîne de caractères permettant à l'utilisateur de sécuriser son compte. Pour être efficace, ce mot de passe doit respecter certaines règles :

- Avoir un nombre de caractères significatif (au moins 8 caractères)
- Posséder au moins un caractère de tous les types (minuscules, majuscules, numériques et caractères spéciaux, comme @, (ou {)
- Ne pas comporter d'informations liées à l'utilisateur (comme son login, le nom de l'entreprise...)
- Ne pas être sous une forme commune (date d'anniversaire, numéro de téléphone...)
- Ne pas être un des mots de passe les plus utilisés (azerty, 1234...)

Ces règles permettent de s'assurer que le mot de passe sera difficilement cassable en utilisant une attaque par force brute, c'est-à-dire en testant toutes les possibilités une à une, ou une attaque par dictionnaire, qui utilise des mots courants.

Fondamental

Le mot de passe ne doit jamais être stocké en clair dans la base de données. Il doit au préalable être *hashé*, c'est-à-dire modifié de manière à ne pas pouvoir être retrouvé en cas de piratage de la base de données. En effet, cela permettrait aux pirates de s'identifier sous n'importe quel compte. Pire, de nombreux utilisateurs utilisent le même mot de passe sur plusieurs sites : les hackers pourraient ainsi tester ces mots de passe sur tous les sites communs (boîtes de messagerie, réseaux sociaux...).

Le hashage diffère du chiffrement : un chiffrement est fait pour être déchiffré lors de son utilisation, tandis que le hashage rend impossible la récupération de la valeur originale.

Nous allons donc devoir stocker un mot de passe hashé dans notre base de données. Selon l'algorithme utilisé, la taille du hash pourra varier. Par exemple, en utilisant l'algorithme BCrypt, le résultat sera sur 60 bits :

```
1 CREATE TABLE users
2 (
3     email VARCHAR(254) NOT NULL UNIQUE,
4     username VARCHAR(20) NOT NULL,
5     password VARCHAR(60) NOT NULL
6 );
```

En PHP, la fonction `password_hash` permet de hasher un mot de passe. Cette fonction prend deux paramètres : le mot de passe à hasher et l'algorithme à utiliser. L'algorithme est précisé par les constantes PHP `PASSWORD_BCRYPT`, `PASSWORD_ARGON2I` et `PASSWORD_ARGON2ID`, les deux derniers n'étant disponibles que si l'extension associée est présente.

Il est également possible de fournir `PASSWORD_DEFAULT` pour laisser PHP décider du meilleur algorithme. Cependant, ce choix étant susceptible de changer entre les versions de PHP, il n'est pas recommandé de l'utiliser.

Au moment de l'inscription, il va donc falloir hasher le mot de passe avant de le stocker en base.

```
1 <?php
2
3 $pdo = new PDO('mysql:host=localhost;dbname=intro_pdo', 'root', '');
4 $statement = $pdo->prepare('INSERT INTO users(email, username, password) VALUES (:email,
5 :username, :password)');
6 $statement->bindValue(':email', 'john@doe.fr');
7
8 // Hash du mot de passe en utilisant BCrypt
9 $statement->bindValue(':password', password_hash('p4$$w0rd', PASSWORD_BCRYPT));
10 if ($statement->execute()) {
11     echo 'L\'utilisateur a bien été créé';
12 } else {
13     echo 'Impossible de créer l\'utilisateur';
14 }
```

Méthode Connexion

Lorsqu'un utilisateur souhaite se connecter, il va renseigner un login et un mot de passe potentiel, qu'il va falloir vérifier. Le but est donc de rechercher un utilisateur ayant le même login que celui fourni et dont le mot de passe hashé correspond au hash de celui fourni. Pour cela, il va falloir utiliser la fonction PHP `password_verify`, qui permet de vérifier qu'un mot de passe correspond à un hash :

```
1 <?php
2
3 $pdo = new PDO('mysql:host=localhost;dbname=intro_pdo', 'root', '');
4 $statement = $pdo->prepare('SELECT * FROM users WHERE email = :email');
5 // On récupère un utilisateur ayant le même login (ici, e-mail)
6 $statement->bindValue(':email', 'john@doe.fr');
7 if ($statement->execute()) {
8     $user = $statement->fetch(PDO::FETCH_ASSOC);
9     if ($user === false) {
10         // Si aucun utilisateur ne correspond au login entré, on affiche une erreur
11         echo 'Identifiants invalides';
12     } else {
13         // On vérifie le hash du password
14         if (password_verify('p4$$w0rd', $user['password'])) {
```

```

15         echo 'Bienvenue ' . $user['username'];
16     } else {
17         echo 'Identifiants invalides';
18     }
19 }
20 } else {
21     echo 'Impossible de récupérer l\'utilisateur';
22 }
23

```

Remarque

Afin d'être vraiment efficace et sécurisée, la fonction `password_hash` ajoute un **salt** au mot de passe, c'est-à-dire une chaîne de caractères générée aléatoirement à chaque appel. Ainsi, chaque hash d'un même mot de passe sera toujours différent. Il n'est donc pas possible de comparer directement le hash stocké en base de données avec le `password_hash` du mot de passe saisi par l'utilisateur au moment de la connexion : même si les mots de passe sont identiques, leurs hash seront différents.

La fonction `password_verify` permet de retrouver le salt et de hasher le mot de passe proposé dans les mêmes conditions que l'original pour comparer leurs valeurs.

Complément **Authentification à deux facteurs**

Pour augmenter la sécurité d'une application, de plus en plus de sites utilisent l'authentification à deux facteurs. Elle permet d'ajouter une étape au couple login/mot de passe en envoyant un code par e-mail ou par numéro de téléphone au moment de la connexion, et de demander à l'utilisateur de le saisir.

C'est une solution très efficace, mais plus coûteuse à mettre en place : elle dépend donc de l'envergure du projet.

Syntaxe **À retenir**

- Un couple login/mot de passe permet de sécuriser une application. Un e-mail est un très bon candidat pour un login.
- Un mot de passe ne doit jamais être stocké en clair, mais utiliser une fonction de hashage : en PHP, on utilise `password_hash`. Au moment de la demande de connexion, on peut utiliser `password_verify` pour vérifier que le mot de passe fourni correspond à un hash.

Complément

Attaque par force brute¹

VII. Exercice : Appliquez la notion**Question**

[solution n°4 p.22]

Une application possède la base d'utilisateurs suivante :

```

1 CREATE TABLE users (
2   email varchar(254) NOT NULL,
3   username varchar(20) UNIQUE NOT NULL,
4   password varchar(60) NOT NULL
5 );
6 INSERT INTO users (email, username, password) VALUES

```

¹ https://fr.wikipedia.org/wiki/Attaque_par_force_brute

```
7 ('dupond@monmail.com', 'rodupond',
  '$2y$10$vEx3REIRj.omkixkE2IMruhroKKA0Lzz/xtL6fFQQ.9irK55DaLsK'),
8 ('jeremy.bratus@monmail.com', 'Jrm',
  '$2y$10$YbXlnPthaF.yjsWZXAnno.q.HU7AQcJP86Q0NxiAtebiPo93ijvLK'),
9 ('john@doe.fr', 'john', '$2y$10$ClqJp6rOm3NehGNxaZbU5e0hZfCJQabcsFpQUWqw5xRk7sQxL9xH0'),
10 ('laure@mondi.com', 'laure', '$2y$10$wbuiouTEJ8FYUYjaG.x5Qe1NGX0VaU61XFqFx0h9LFqvqIwHu2qbS'),
11 ('sarah@monmail.com', 'Saraaa',
  '$2y$10$0ErjVwZeJLjNlyYh4l2m9eRJYLGGokpXwVvl9C0tW4EM8NQw0DJ8y'),
12 ('sophie@granf.com', 'soso', '$2y$10$7hGETxVUDb.UWSdq8HkKw.07wMskSi9tZmPEMujzDxjPPOYCsLuuu2');
```

Une activité suspecte a été détectée sur le site : des hackers semblent tester le mot de passe `ch4t0n!` en attaquant par force brute les noms d'utilisateurs. Si des utilisateurs utilisent ce mot de passe, la sécurité de leurs comptes risque d'être compromise.

Écrivez un script permettant de lister les e-mails de tous les utilisateurs possédant ce mot de passe afin que l'équipe d'assistance puisse les contacter pour qu'ils le changent.

Indice :

Il va falloir récupérer tous les utilisateurs et comparer tous les mots de passe à `ch4t0n!`.

VIII. Les rôles

Objectifs

- Comprendre la notion de rôle
- Gérer les droits des utilisateurs

Mise en situation

Tous les utilisateurs de l'application n'ont pas forcément accès aux mêmes pages ou aux mêmes fonctionnalités. Par exemple, un utilisateur non connecté aura souvent accès à moins de choses qu'un administrateur. Pour gérer ces différents cas, on utilise des **rôles**.

Définition Les rôles

Un rôle peut être assimilé à un groupe auquel appartient à l'utilisateur. Tous les membres d'un même groupe disposent d'accès à des pages ou à des fonctionnalités différentes. Un utilisateur peut avoir un ou plusieurs rôles. Chaque rôle est représenté par une chaîne de caractères.

On peut imaginer, par exemple, qu'un groupe `utilisateurs` a le droit de visualiser les articles d'un blog, tandis que le groupe `administrateurs` a le droit d'en ajouter ou d'en supprimer.

On peut distinguer deux types de rôles : les rôles configurés dans l'application, paramétrés directement en base de données, et ceux définis par le métier, qui sont des rôles déterminés par des règles métier. Dans les deux cas, le but final est d'affecter une liste de rôles à nos utilisateurs au moment de la connexion. Pour cela, notre application comportera une classe `User` qui contiendra les propriétés de notre utilisateur, ainsi qu'un tableau de rôles.

```
1 <?php
2
3 class User
4 {
5     private string $id;
6     private string $email;
7     private string $password;
8     // Tableau de rôles
9     private array $roles = [];
10
11     public function getId(): string
```

```

12 {
13     return $this->id;
14 }
15
16 public function addRole(string $role): void
17 {
18     $this->roles[] = $role;
19 }
20
21 public function getRoles(): array
22 {
23     return $this->roles;
24 }
25 }

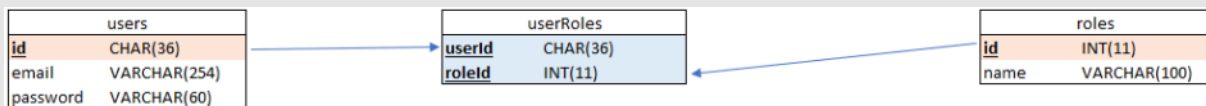
```

Remarque

Il existe beaucoup de manières différentes de gérer les rôles dans une application, selon les besoins du projet. Dans ce chapitre, nous allons voir un exemple simple qui se rapproche de la gestion des utilisateurs telle qu'elle est implémentée dans le framework Symfony, mais c'est loin d'être la seule implémentation qui existe. N'hésitez pas à adapter ce système selon les cas.

Méthode Rôles configurés dans l'application

Il est possible de gérer des rôles des utilisateurs en les configurant directement en base de données. Pour cela, en plus de la table des utilisateurs, une table `roles` va contenir tous les rôles de l'application, et une table `userRoles` attribuera des rôles à chaque utilisateur.



Le code permettant de générer ce schéma est le suivant :

```

1 CREATE TABLE users
2 (
3     id CHAR(36) NOT NULL PRIMARY KEY,
4     email VARCHAR(254) NOT NULL UNIQUE,
5     password VARCHAR(60) NOT NULL
6 );
7
8 CREATE TABLE roles
9 (
10    id INT(11) NOT NULL PRIMARY KEY,
11    name VARCHAR(100) NOT NULL UNIQUE
12 );
13
14 CREATE TABLE userRoles
15 (
16    userId CHAR(36) NOT NULL,
17    roleId INT(11) NOT NULL,
18    PRIMARY KEY (userId, roleId),
19    FOREIGN KEY (userId) REFERENCES users(id),
20    FOREIGN KEY (roleId) REFERENCES roles(id)
21 );

```

Pour attribuer un rôle à un utilisateur, il suffit de créer une ligne de données dans la table `userRoles`.

Pour récupérer la liste des rôles d'un utilisateur, il suffit d'effectuer une jointure entre la table `userRoles` et la table `roles` une fois la connexion effectuée :

```
1 <?php
2 // ... Processus de connexion
3 // $user est un objet User retourné par PDO
4
5 $statement = $pdo->prepare('SELECT * FROM userRoles JOIN roles ON roles.id = userRoles.roleId
6 WHERE id = :id');
7 $statement->bindValue(':id', $user->getId());
8 if ($statement->execute()) {
9     while ($role = $statement->fetch(PDO::FETCH_ASSOC)) {
10         $user->addRole($role['name']);
11     }
12 }
```

Remarque

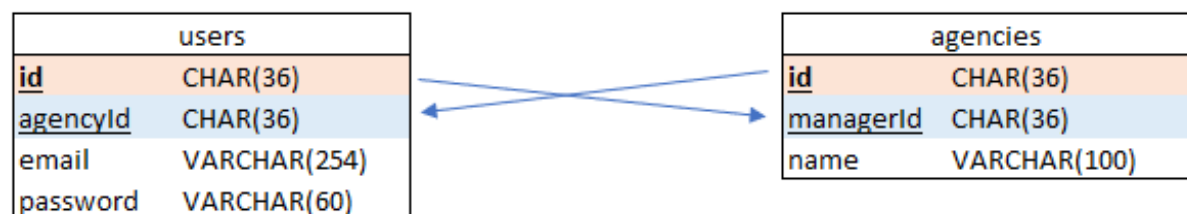
La clé primaire de la table `userRoles` est composée de deux colonnes : `userId` et `userRole`. Cela signifie qu'un même utilisateur ne pourra pas avoir plusieurs fois le même rôle.

Méthode Rôles définis par le métier

Il peut arriver que les rôles soient déduits depuis des règles métier plutôt que d'être configurés dans la base de données. Dans ce cas, plutôt que d'effectuer un appel en base, il va falloir vérifier toutes les règles métier et remplir le tableau de rôles en conséquence.

Exemple

Prenons l'exemple d'un réseau d'agences immobilières qui utilise une application de gestion de biens. Chaque agence possède un ensemble d'utilisateurs, ainsi qu'un chef d'agence, également utilisateur du logiciel. La table `agencies` aura donc la structure suivante :



Pour déterminer si un utilisateur possède le rôle de chef d'agence ou non, il faut vérifier si son identifiant est présent dans un champ `managerId`. Une façon simple et performante de faire cela est de compter le nombre d'agences ayant l'utilisateur comme manager :

```
1 <?php
2 // ... Processus de connexion
3 // $user est un objet User retourné par PDO
4
5 $statement = $pdo->prepare('SELECT COUNT(*) AS count FROM agencies WHERE managerId = :id');
6 $statement->bindValue(':id', $user->getId());
7 if ($statement->execute()) {
8     $count = $statement->fetch(PDO::FETCH_ASSOC);
9     if ($count['count'] > 0) {
10         // Si l'utilisateur est chef d'une agence, alors on lui ajoute le rôle ROLE_MANAGER
11         $user->addRole('ROLE_MANAGER');
12     }
13 }
```

13 }

Méthode Vérifier les droits

Une fois les rôles de notre utilisateur déterminés, il est possible de limiter certaines fonctionnalités de l'application à certains rôles. Pour cela, il suffit de comparer le rôle requis à la liste de l'utilisateur connecté. Il est possible d'utiliser la fonction PHP `in_array`, qui permet de savoir si un élément se trouve dans un tableau :

```
1 <?php
2
3 // ... Processus de connexion et de deduction des rôles
4
5 if (!in_array('ROLE_MANAGER', $user->getRoles()))
6 {
7     throw new Exception('Vous n\'avez pas le droit d\'accéder à cette fonctionnalité');
8 }
```

Syntaxe À retenir

- Un rôle permet de définir les droits d'un utilisateur.
- Ils peuvent être configurés en base de données ou déterminés selon des règles métier.
- Au moment de la connexion, il est possible de remplir le tableau de rôles de l'utilisateur.
- Pour restreindre une fonctionnalité à certains rôles, il suffit de comparer le tableau au rôle requis grâce à la fonction `in_array`.

Complément`in_array1`**IX. Exercice : Appliquez la notion****Question**

[solution n°5 p.22]

Une entreprise de vente d'outils de jardinage dispose d'une application permettant de gérer ses vendeurs et ses ventes (dont le prix est stocké en **centimes**). Le schéma de la base de données est celui-ci :

```
1 CREATE TABLE users (
2     id CHAR(36) NOT NULL PRIMARY KEY,
3     email VARCHAR(254) UNIQUE NOT NULL,
4     username VARCHAR(20) NOT NULL,
5     password VARCHAR(60) NOT NULL
6 );
7
8 CREATE TABLE sales (
9     id CHAR(36) NOT NULL PRIMARY KEY,
10    idUser CHAR(36) NOT NULL,
11    price INT(11) NOT NULL
12 );
13
14 INSERT INTO users (id, email, username, password) VALUES
15 ('2a2b8dc9-cc29-11ea-b05e-1831bf21e339', 'dupond@monmail.com', 'rodupond',
    '$2y$10$VEx3REIRj.omkixkE2IMruhroKKA0Lzz/xtL6fFQQ.9irK55DaLsK'),
```

1 <https://www.php.net/manual/fr/function.in-array.php>

```

16 ('2a2b9b27-cc29-11ea-b05e-1831bf21e339', 'laure@mondi.com', 'laure',
17 '52y$10$wbiuoUtEJ8FYUYjaG.x5Qe1NGX0VaU61XFqFx0h9LFqvqIwHu2qbS');
18 INSERT INTO sales (id, idUser, price) VALUES
19 (UUID(), '2a2b8dc9-cc29-11ea-b05e-1831bf21e339', 10000),
20 (UUID(), '2a2b8dc9-cc29-11ea-b05e-1831bf21e339', 20000),
21 (UUID(), '2a2b8dc9-cc29-11ea-b05e-1831bf21e339', 1000),
22 (UUID(), '2a2b9b27-cc29-11ea-b05e-1831bf21e339', 1000),
23 (UUID(), '2a2b8dc9-cc29-11ea-b05e-1831bf21e339', 32000),
24 (UUID(), '2a2b8dc9-cc29-11ea-b05e-1831bf21e339', 64000),
25 (UUID(), '2a2b9b27-cc29-11ea-b05e-1831bf21e339', 12000),
26 (UUID(), '2a2b9b27-cc29-11ea-b05e-1831bf21e339', 13000);

```

L'entreprise voudrait ajouter une nouvelle page dans l'application, qui ne serait accessible que par les vendeurs « senior », c'est-à-dire aux vendeurs qui ont vendu pour plus de 1000 € de produits dans leur carrière. Pour vous aider, voici le script permettant de connecter un utilisateur :

User.php :

```

1 <?php
2
3 class User
4 {
5     private string $id;
6     private string $username;
7     private string $password;
8     private string $email;
9
10    // Méthode statique permettant de retourner un User a partir d'un e-mail et d'un mot de
11    passe
12    public static function connect(PDO $pdo, string $email, string $password): User
13    {
14        $statement = $pdo->prepare('SELECT * FROM users WHERE email = :email');
15        $statement->setFetchMode(PDO::FETCH_CLASS, 'User');
16        $statement->bindValue(':email', $email);
17        if ($statement->execute()) {
18            while ($user = $statement->fetch()) {
19                if ($user->isPasswordValid($password)) {
20                    return $user;
21                }
22            }
23
24            throw new Exception('Identifiants invalides');
25        }
26
27        public function isPasswordValid(string $password): bool
28        {
29            return password_verify($password, $this->password);
30        }
31    }

```

index.php :

```

1 <?php
2
3 require_once 'User.php';
4
5 $pdo = new PDO('mysql:host=localhost;dbname=intro_pdo', 'root', '');
6 // Pour se connecter en tant que Robert Dupont (vendeur "senior")
7 $user = User::connect($pdo, 'dupond@monmail.com', 'r0r0dup0nt!');

```



```
8
9 // Pour se connecter en tant que Laure Mondì (vendeuse "junior")
10 //$user = User::connect($pdo, 'laure@mondi.com', 'ch4t0n!');
```

Modifiez le code de manière à calculer le rôle de l'utilisateur (ROLE_JUNIOR ou ROLE_SENIOR) au moment de la connexion, puis complétez la page **index.php** pour qu'elle ne soit accessible que par les vendeurs senior.

X. Essentiel

XI. Auto-évaluation

A. Exercice final

Exercice 1

[solution n°6 p.24]

Exercice

Parmi ces données, lesquelles sont des données métier ?

- ☐ L'identifiant technique
- ☐ Le login
- ☐ Le numéro de carte bleue
- ☐ Le prénom
- ☐ Le mot de passe

Exercice

Vérifier qu'un numéro de carte bleue ne soit composé que de chiffres sous la forme XXXX-XXXX-XXXX-XXXX suffit à le valider.

- ☐ Vrai
- ☐ Faux

Exercice

Parmi ces types de données, lesquels peuvent être utilisés pour stocker l'identifiant technique d'un utilisateur ?

- ☐ CHAR(36)
- ☐ VARCHAR(12)
- ☐ UUID
- ☐ INTEGER(11)

Exercice

Il y a plus de chances qu'il y ait deux fois les mêmes tirages de loto que d'avoir une duplication d'UUID dans une base de données.

- ☐ Vrai
- ☐ Faux

Exercice

Quels sont les inconvénients d'un entier auto-incrémenté en tant qu'identifiant technique ?

- ☐ Ils sont prévisibles
- ☐ Ils peuvent créer des conflits en cas de fusion de bases de données
- ☐ Ils ne sont pas performants
- ☐ Tous les SGBD ne supportent pas l'auto-incrémentation
- ☐ Ils peuvent donner des indications sur le contenu de la table

Exercice

Parmi ces données, laquelle serait le meilleur choix en tant que login ?

- ☐ Le nom de famille de l'utilisateur
- ☐ L'identifiant technique
- ☐ L'adresse e-mail
- ☐ Le mot de passe

Exercice

Quels sont les critères pour avoir un mot de passe efficace contre les attaques ?

- ☐ Être facile à retenir
- ☐ Être composé de beaucoup de caractères
- ☐ Posséder un mélange de minuscules, de majuscules, de nombres et de symboles
- ☐ Avoir le nom de l'utilisateur dans le mot de passe
- ☐ Ne pas avoir de mot de passe : personne ne pensera à essayer ça !

Exercice

Comment peut-on stocker les mots de passe en base de données ?

- ☐ Sous forme de chaîne de 8 caractères maximum
- ☐ En hashant le mot de passe
- ☐ Si la base de données est suffisamment sécurisée, on peut les stocker tels quels
- ☐ En chiffrant le mot de passe

Exercice

Comment trouver un utilisateur à partir de son login et de son mot de passe ?

- ☐ `SELECT * FROM users WHERE login = :login`, puis on utilise la fonction `password_verify` sur le résultat
- ☐ `SELECT * FROM users WHERE login = :login AND password = :password;`, **:password** étant le mot de passe proposé
- ☐ `SELECT * FROM users WHERE login = :login AND password = :password;`, **:password** étant le mot de passe proposé hashé avec la fonction `password_hash`

Exercice

À quoi servent les rôles ?

- ☐ Définir des groupes afin de construire différentes communautés dans une application
- ☐ Mettre des étiquettes sur les utilisateurs pour les retrouver plus facilement en base de données
- ☐ Définir les droits et les accès de chaque utilisateur

B. Exercice : Défi

Il est temps de mettre en commun tout ce qu'on a vu dans ce cours pour créer un formulaire d'inscription et de connexion.

Question 1

[solution n°7 p.26]

Un restaurant veut proposer un système de réservation en ligne à ses clients. Pour cela, il aimerait ajouter des fonctionnalités d'inscription et de connexion sur son site.

Le restaurateur vous laisse gérer la structure de la table comme vous le souhaitez : les seules informations qui l'intéressent sont l'e-mail, le nom, le prénom et le numéro de département de leurs clients, mais vous pouvez rajouter autant de champs que nécessaires pour le système de connexion.

En revanche, il souhaiterait rester discret sur le nombre d'utilisateurs de sa plateforme : il ne faut pas qu'un concurrent puisse déduire le nombre d'utilisateurs inscrits.

Avec ces informations, créez une base de données d'utilisateurs, puis créez une classe PHP permettant de les gérer.

Indice :

L'e-mail est un bon candidat pour le login, il est inutile d'en rajouter un.

Indice :

Attention : les numéros de département ne sont pas tous des nombres (2A et 2B pour la Corse), et ne sont pas tous sur deux caractères non plus (971 pour la Guadeloupe, par exemple).

Indice :

Contrairement aux auto-incréments, les UUID permettent de ne pas exposer le volume de données contenues dans une base.

Question 2

[solution n°8 p.27]

Créez un `UserManager` qui va vous permettre de manipuler les utilisateurs en base de données. Cette classe devra comporter une méthode `subscribe` prenant en paramètres toutes les informations nécessaires à la création d'un utilisateur et qui retourne **vrai** si l'utilisateur a pu être inséré en base de données, **faux** sinon. Pour cela, une instance de PDO devra être fournie au manager par injection de dépendances.

Pour simplifier la question, la seule vérification à réaliser sur le mot de passe doit être sa taille, supérieure à 7 caractères. Il ne sera pas nécessaire de vérifier les autres données. Le mot de passe devra être hashé en utilisant l'algorithme BCrypt.

Créez ensuite un utilisateur de votre choix dans le fichier **index.php**.

Indice :

Pour le moment, il n'est pas utile d'utiliser la classe `User`.

Question 3

[solution n°9 p.28]

Ajoutez une méthode `connect` à votre `userManager`. Cette méthode doit prendre en paramètres un login et un mot de passe et doit retourner le `User` correspondant. Si aucun utilisateur n'est trouvé, alors cette méthode doit lancer une exception.

Connectez-vous ensuite en tant que l'utilisateur précédemment créé pour tester votre fonction : affichez *Bonjour*, suivi de son nom et de son prénom. Pour cela, créez une méthode `sayHello` dans la classe `User` permettant de retourner la phrase à afficher.

Indice :

N'oubliez pas d'inclure la classe `User` là où vous en avez besoin.

Question 4

[solution n°10 p.29]

Le restaurant souhaite également proposer à ses clients un service de plats en livraison. Cependant, le restaurant n'est pas capable de livrer dans toute la France, donc cette fonctionnalité doit être limitée aux départements 75, 94, 92 et 93.

Modifiez votre code de manière à ce que seuls les clients de ces départements aient le droit d'accéder à la page du service de livraison.

Indice :

Le but de l'exercice est de créer un rôle (`ROLE_DELIVERABLE`, par exemple) et de le déterminer au moment de la connexion.

Indice :

La fonction `in_array` peut également être utile pour gérer les départements.

Solutions des exercices

p.5 Solution n°1

La vérification se fait en deux étapes : une expression régulière vérifie que le format est bien le bon, puis un calcul vérifie la clé de contrôle.

```

1 <?php
2
3 function isSocialNumberValid(string $number): bool
4 {
5     // On vérifie le format
6     if (!preg_match('/[12][0-9]{2}(0[1-9]|1[0-2])[0-9]{10}/', $number)) {
7         return false;
8     }
9
10    // On vérifie la clef de contrôle
11    $firstNumbers = (int)substr($number, 0, -2);
12    $controlKey = (int)substr($number, -2);
13
14    return (97 - ($firstNumbers % 97)) === $controlKey;
15 }
16
17 var_dump(isSocialNumberValid('369054958815780')); // false
18 var_dump(isSocialNumberValid('269134958815780')); // false
19 var_dump(isSocialNumberValid('26914958815780')); // false
20 var_dump(isSocialNumberValid('269054958815781')); // false
21 var_dump(isSocialNumberValid('269054958815780')); // true

```

Décomposons l'expression :

- `[12]` permet de vérifier que le premier chiffre est soit 1, soit 2
- `[0-9]{2}` permet de vérifier que l'année est bien comprise entre 00 et 99
- `(0[1-9]|1[0-2])` permet de vérifier que le mois est bien compris entre 01 et 12. Elle est composée de deux alternatives : `0[1-9]` qui vérifie les mois de 01 à 09, et `1[0-2]` qui permet de vérifier les mois de 10 à 12
- `[0-9]{10}` permet de vérifier que tous les autres nombres sont compris entre 0 et 9

Il n'est pas utile de vérifier que la clé est comprise entre 00 et 97 dans la regex, puisqu'on le vérifie ensuite.

p.8 Solution n°2

Le fait que l'entreprise ait pour but à long terme de fusionner les informations des différentes bases de données indique que nous allons devoir utiliser des UUID en clé primaire. En effet, en utilisant des entiers auto-incrémentés, il va y avoir plusieurs véhicules n°1 répartis sur les différentes bases. Au moment de la fusion, il va donc falloir faire un lourd processus de changement des identifiants.

```

1 CREATE TABLE vehicles
2 (
3     -- MariaDB ne possède pas de type UUID, donc l'identifiant est un CHAR(36)
4     id CHAR(36) NOT NULL PRIMARY KEY,
5     licensePlate VARCHAR(10) NOT NULL,
6     brand VARCHAR(100) NOT NULL
7 );
8
9 CREATE TABLE drivers
10 (
11     id CHAR(36) NOT NULL PRIMARY KEY,

```

```

12     firstName VARCHAR(100) NOT NULL,
13     lastName VARCHAR(100) NOT NULL
14 );
15
16 CREATE TABLE trips
17 (
18     id CHAR(36) NOT NULL PRIMARY KEY,
19     vehicleId CHAR(36) NOT NULL,
20     driverId CHAR(36) NOT NULL,
21     date DATETIME NOT NULL,
22     FOREIGN KEY (vehicleId) REFERENCES vehicles(id),
23     FOREIGN KEY (driverId) REFERENCES drivers(id)
24 );
25

```

p. 8 Solution n°3

```

1 INSERT INTO vehicles VALUES (UUID(), 'XX-123-XX', 'SuperBus');
2 INSERT INTO drivers VALUES (UUID(), 'John', 'Doe');

```

p. 11 Solution n°4

```

1 <?php
2
3 $pdo = new PDO('mysql:host=localhost;dbname=intro_pdo', 'root', '');
4 // On récupère tous les utilisateurs
5 $statement = $pdo->prepare('SELECT * FROM users');
6 if ($statement->execute()) {
7     while ($user = $statement->fetch(PDO::FETCH_ASSOC)) {
8         // Pour tous les utilisateurs, on vérifie si leur mot de passe correspond à celui
9         compromis
10         if (password_verify('ch4t0n!', $user['password'])) {
11             echo $user['email'].'<br>';
12         }
13     }
14 } else {
15     echo 'Impossible de récupérer la liste des utilisateurs';
16 }

```

Grâce à ce script, l'équipe d'assistance va pouvoir envoyer un e-mail aux adresses **jeremy.bratus@monmail.com** et **laure@mondi.com** pour les prévenir de changer leur mot de passe.

p. 15 Solution n°5

User.php :

```

1 <?php
2
3 class User
4 {
5     private string $id;
6     private string $username;
7     private string $password;
8     private string $email;
9     private array $roles = [];

```

```

10
11     public static function connect(PDO $pdo, string $email, string $password): User
12     {
13         $statement = $pdo->prepare('SELECT * FROM users WHERE email = :email');
14         $statement->setFetchMode(PDO::FETCH_CLASS, 'User');
15         $statement->bindValue(':email', $email);
16         if ($statement->execute()) {
17             while ($user = $statement->fetch()) {
18                 if ($user->isPasswordValid($password)) {
19                     // Si le mot de passe est valide, alors on récupère la somme des ventes
20 de l'utilisateur
21                 $sumStatement = $pdo->prepare('SELECT SUM(price) AS sum FROM sales WHERE
22 idUser = :id');
23                 $sumStatement->bindValue(':id', $user->getId());
24                 if ($sumStatement->execute()) {
25                     $sum = $sumStatement->fetch(PDO::FETCH_ASSOC);
26                     // Si la somme est supérieure à 1000€ (donc 100000 centimes), on
27 attribue le rôle de senior.
28                     if ($sum['sum'] > 100000) {
29                         $user->addRole('ROLE_SENIOR');
30                     } else {
31                         $user->addRole('ROLE_JUNIOR');
32                     }
33                 } else {
34                     throw new Exception('Erreur lors de la récupération des ventes');
35                 }
36                 return $user;
37             }
38         }
39         throw new Exception('Identifiants invalides');
40     }
41
42     public function isPasswordValid(string $password): bool
43     {
44         return password_verify($password, $this->password);
45     }
46
47     public function addRole(string $role): void
48     {
49         $this->roles[] = $role;
50     }
51
52     public function getRoles(): array
53     {
54         return $this->roles;
55     }
56
57     public function getId(): string
58     {
59         return $this->id;
60     }
61 }

```

index.php :


```
1 <?php
2
3 require_once 'User.php';
4
5 $pdo = new PDO('mysql:host=localhost;dbname=intro_pdo', 'root', '');
6 $user = User::connect($pdo, 'dupond@monmail.com', 'r0r0dmp0nt');
7 // $user = User::connect($pdo, 'laure@mondi.com', 'ch4t0n!');
8
9 if (!in_array('ROLE_SENIOR', $user->getRoles())) {
10     echo 'Accès interdit';
11 } else {
12     echo 'Bonjour vendeur senior !';
13 }
```

Exercice p. 17 Solution n°6

Exercice

Parmi ces données, lesquelles sont des données métier ?


- ☐ L'identifiant technique
- ☐ Le login
- ☒ Le numéro de carte bleue
- ☒ Le prénom
- ☐ Le mot de passe

 Le prénom et le numéro de carte bleue sont des données métier : elle ne servent pas à gérer les utilisateurs et dépendent de l'application.

Exercice

Vérifier qu'un numéro de carte bleue ne soit composé que de chiffres sous la forme XXXX-XXXX-XXXX-XXXX suffit à le valider.


- ☐ Vrai
- ☒ Faux

 Faux : il faut au moins implémenter l'algorithme de Luhn pour s'assurer de la validité du numéro de carte. Si c'est possible, on peut également vérifier l'identifiant de la banque.

Exercice

Parmi ces types de données, lesquels peuvent être utilisés pour stocker l'identifiant technique d'un utilisateur ?


- ☒ CHAR(36)
- ☐ VARCHAR(12)
- ☒ UUID
- ☒ INTEGER(11)

-  Il est possible de stocker un identifiant technique sous forme d'entier auto-incrémenté ou d'UUID. Selon les moteurs de base de données, l'UUID pourra soit posséder son propre type, soit être stocké sous la forme d'une chaîne de caractères de taille 36.

Exercice

Il y a plus de chances qu'il y ait deux fois les mêmes tirages de loto que d'avoir une duplication d'UUID dans une base de données.


- ☒ Vrai
- ☐ Faux

-  Il existe 2^{112} possibilités pour générer un UUID. Avec ses presque 2 millions de possibilités pour un tirage, il y a bien plus de chances d'avoir une duplication au loto qu'en base de données.

Exercice


Quels sont les inconvénients d'un entier auto-incrémenté en tant qu'identifiant technique ?

- ☒ Ils sont prévisibles
- ☒ Ils peuvent créer des conflits en cas de fusion de bases de données
- ☐ Ils ne sont pas performants
- ☐ Tous les SGBD ne supportent pas l'auto-incrémentation
- ☒ Ils peuvent donner des indications sur le contenu de la table

-  Les entiers auto-incrémentés sont prévisibles et ne sont pas uniques entre les bases de données, ce qui peut provoquer des conflits en cas de fusion de bases. De plus, un utilisateur peut déduire le nombre d'éléments dans une table en regardant l'identifiant généré.

Exercice

Parmi ces données, laquelle serait le meilleur choix en tant que login ?

- ☐ Le nom de famille de l'utilisateur
 - ☐ L'identifiant technique
 - ☒ L'adresse e-mail
 - ☐ Le mot de passe
-  Une adresse e-mail est unique et facile à mémoriser pour l'utilisateur : c'est le candidat idéal.

Exercice

Quels sont les critères pour avoir un mot de passe efficace contre les attaques ?

- ☐ Être facile à retenir
- ☒ Être composé de beaucoup de caractères
- ☒ Posséder un mélange de minuscules, de majuscules, de nombres et de symboles
- ☐ Avoir le nom de l'utilisateur dans le mot de passe
- ☐ Ne pas avoir de mot de passe : personne ne pensera à essayer ça !

- Q Un mot de passe doit faire au moins 8 caractères et être composé de minuscules, majuscules, nombres et symboles pour être efficace. Il faut éviter les mots courants, les noms d'utilisateurs, les dates d'anniversaire et les mots de passe courants. Bien évidemment, le mot de passe est obligatoire.

Exercice

Comment peut-on stocker les mots de passe en base de données ?

- ☐ Sous forme de chaîne de 8 caractères maximum
- ☒ En hashant le mot de passe
- ☐ Si la base de données est suffisamment sécurisée, on peut les stocker tels quels
- ☐ En chiffrant le mot de passe

- Q Quel que soit le niveau de sécurité de la base de données, il est nécessaire de hasher le mot de passe. Il ne faut pas confondre hashage et chiffrement : un chiffrement est fait pour être déchiffré lors de son utilisation, tandis que le hashage rend impossible la récupération de la valeur originale.

Exercice

Comment trouver un utilisateur à partir de son login et de son mot de passe ?

- ☒ `SELECT * FROM users WHERE login = :login`, puis on utilise la fonction `password_verify` sur le résultat
 - ☐ `SELECT * FROM users WHERE login = :login AND password = :password;`, **:password** étant le mot de passe proposé
 - ☐ `SELECT * FROM users WHERE login = :login AND password = :password;`, **:password** étant le mot de passe proposé hashé avec la fonction `password_hash`
- Q Si un même mot de passe est hashé plusieurs fois, alors un résultat différent sera produit à chaque fois. Il n'est donc pas possible de vérifier un mot de passe en le comparant directement avec la valeur stockée en base de données. Il faut utiliser la fonction `password_verify`.

Exercice

À quoi servent les rôles ?

- ☐ Définir des groupes afin de construire différentes communautés dans une application
 - ☐ Mettre des étiquettes sur les utilisateurs pour les retrouver plus facilement en base de données
 - ☒ Définir les droits et les accès de chaque utilisateur
- Q Les rôles permettent d'interdire ou d'autoriser l'accès à certaines fonctionnalités de l'application.

p. 19 Solution n°7

Le schéma de base de données est le suivant :

```
1 CREATE TABLE users (
2   id CHAR(36) NOT NULL PRIMARY KEY,
3   email VARCHAR(254) UNIQUE NOT NULL,
4   password VARCHAR(60) NOT NULL,
5   firstName VARCHAR(100) NOT NULL,
6   lastName VARCHAR(100) NOT NULL,
7   department VARCHAR(3) NOT NULL
8 );
```

Fichier **User.php** :

```
1 <?php
2
3 class User
4 {
5     private string $id;
6     private string $email;
7     private string $password;
8     private string $firstName;
9     private string $lastName;
10    private string $department;
11 }
```

p. 19 Solution n°8

Fichier **userManager.php** :

```
1 <?php
2
3 class UserManager
4 {
5     private PDO $pdo;
6
7     public function __construct(PDO $pdo)
8     {
9         $this->pdo = $pdo;
10    }
11
12    public function subscribe(string $email, string $password, string $firstName, string
13    $lastName, string $department): bool
14    {
15        // Vérification de la taille du mot de passe
16        if (strlen($password) < 8) {
17            // Si le mot de passe n'est pas au bon format, on retourne false
18            return false;
19        }
20
21        // Préparation de la requête
22        $statement = $this->pdo->prepare('
23    INSERT INTO users (id, email, password, firstName, lastName, department)
24    VALUES (UUID(), :email, :password, :firstName, :lastName, :department)'
25        );
26
27        // Injection des valeurs des marqueurs
28        $statement->bindValue(':email', $email, PDO::PARAM_STR);
29        // Hashage du mot de passe
30        $statement->bindValue(':password', password_hash($password, PASSWORD_BCRYPT),
31        PDO::PARAM_STR);
32        $statement->bindValue(':firstName', $firstName, PDO::PARAM_STR);
33        $statement->bindValue(':lastName', $lastName, PDO::PARAM_STR);
34        $statement->bindValue(':department', $department, PDO::PARAM_STR);
35
36        // On retourne le statut d'exécution de la requête
37        return $statement->execute();
38    }
39 }
```

Fichier **index.php** :

```
1 <?php
2
3 require_once 'userManager.php';
4
5 $pdo = new PDO('mysql:host=localhost;dbname=restaurant', 'root', '');
6 // Injection de l'objet PDO dans le manager
7 $manager = new UserManager($pdo);
8 if ($manager->subscribe('john@doe.com', 'p4$$w0rd', 'John', 'Doe', '2A')) {
9     echo 'Inscription réussie';
10 } else {
11     echo 'Echec lors de l\'inscription';
12 }
```

p. 19 Solution n°9

Fichier **User.php** :

```
1 <?php
2
3 class User
4 {
5     private string $id;
6     private string $email;
7     private string $password;
8     private string $firstName;
9     private string $lastName;
10    private string $department;
11
12    public function getPassword(): string
13    {
14        return $this->password;
15    }
16
17    public function sayHello(): string
18    {
19        return 'Bonjour ' . $this->firstName . ' ' . $this->lastName;
20    }
21 }
```

Fichier **UserManager.php** :

```
1 <?php
2
3
4 class UserManager
5 {
6     private PDO $pdo;
7
8     public function __construct(PDO $pdo)
9     {
10         $this->pdo = $pdo;
11     }
12
13     public function subscribe(string $email, string $password, string $firstName, string
14     $lastName, string $department): bool
15     {
16         if (strlen($password) < 8) {
```

```

16         return false;
17     }
18
19     $statement = $this->pdo->prepare('
20 INSERT INTO users (id, email, password, firstName, lastName, department)
21 VALUES (UUID(), :email, :password, :firstName, :lastName, :department)'
22 );
23 $statement->bindValue(':email', $email, PDO::PARAM_STR);
24 $statement->bindValue(':password', password_hash($password, PASSWORD_BCRYPT),
PDO::PARAM_STR);
25 $statement->bindValue(':firstName', $firstName, PDO::PARAM_STR);
26 $statement->bindValue(':lastName', $lastName, PDO::PARAM_STR);
27 $statement->bindValue(':department', $department, PDO::PARAM_STR);
28
29     return $statement->execute();
30 }
31
32 public function connect(string $email, string $password): User
33 {
34     require_once 'User.php';
35
36     // On cherche un utilisateur ayant l'adresse e-mail correspondante
37     $statement = $this->pdo->prepare('SELECT * FROM users WHERE email = :email');
38     $statement->setFetchMode(PDO::FETCH_CLASS, 'User');
39     $statement->bindValue(':email', $email);
40     if ($statement->execute()) {
41         // Il y a une contrainte d'unicité sur l'e-mail : il n'est donc pas utile de
faire une boucle
42         $user = $statement->fetch();
43         // Il faut vérifier que fetch n'a pas retourné false avant de vérifier le mot de
44 passe
         if ($user !== false && password_verify($password, $user->getPassword())) {
45             return $user;
46         }
47     }
48
49     throw new Exception('Identifiants invalides');
50 }
51 }

```

Fichier **index.php** :

```

1 <?php
2
3 require_once 'UserManager.php';
4
5 $pdo = new PDO('mysql:host=localhost;dbname=restaurant', 'root', '');
6 $manager = new UserManager($pdo);
7 $user = $manager->connect('john@doe.com', 'p4$w0rd');
8 echo $user->sayHello();

```

Fichier **User.php** :

```

1 <?php
2
3 class User
4 {
5     private string $id;
6     private string $email;
7     private string $password;
8     private string $firstName;
9     private string $lastName;
10    private string $department;
11    private array $roles = [];
12
13    public function getPassword(): string
14    {
15        return $this->password;
16    }
17
18    public function getDepartment(): string
19    {
20        return $this->department;
21    }
22
23    public function sayHello(): string
24    {
25        return 'Bonjour ' . $this->firstName . ' ' . $this->lastName;
26    }
27
28    public function addRole(string $role): void
29    {
30        $this->roles[] = $role;
31    }
32
33    public function getRoles(): array
34    {
35        return $this->roles;
36    }
37 }

```

Fichier **UserManager.php** :

```

1 <?php
2
3
4 class UserManager
5 {
6     private PDO $pdo;
7
8     public function __construct(PDO $pdo)
9     {
10        $this->pdo = $pdo;
11    }
12
13    public function subscribe(string $email, string $password, string $firstName, string
14    $lastName, string $department): bool
15    {
16        if (strlen($password) < 8) {
17            // Si le mot de passe n'est pas au bon format, on retourne false
18            return false;
19        }
20    }
21 }

```

```

18     }
19
20     $statement = $this->pdo->prepare('
21 INSERT INTO users (id, email, password, firstName, lastName, department)
22 VALUES (UUID(), :email, :password, :firstName, :lastName, :department)'
23 );
24 $statement->bindValue(':email', $email, PDO::PARAM_STR);
25 $statement->bindValue(':password', password_hash($password, PASSWORD_BCRYPT),
PDO::PARAM_STR);
26 $statement->bindValue(':firstName', $firstName, PDO::PARAM_STR);
27 $statement->bindValue(':lastName', $lastName, PDO::PARAM_STR);
28 $statement->bindValue(':department', $department, PDO::PARAM_STR);
29
30     return $statement->execute();
31 }
32
33 public function connect(string $email, string $password): User
34 {
35     require_once 'User.php';
36
37     $statement = $this->pdo->prepare('SELECT * FROM users WHERE email = :email');
38     $statement->setFetchMode(PDO::FETCH_CLASS, 'User');
39     $statement->bindValue(':email', $email);
40     if ($statement->execute()) {
41         $user = $statement->fetch();
42         if ($user !== false && password_verify($password, $user->getPassword())) {
43             // On vérifie si le département de l'utilisateur est dans la liste des
départements autorisés
44             if (in_array($user->getDepartment(), ['75', '94', '92', '93'])) {
45                 $user->addRole('ROLE_DELIVERABLE');
46             }
47
48             return $user;
49         }
50     }
51
52     throw new Exception('Identifiants invalides');
53 }
54 }

```

Fichier **index.php** :

```

1 <?php
2
3 require_once 'UserManager.php';
4
5 $pdo = new PDO('mysql:host=localhost;dbname=restaurant', 'root', '');
6 $manager = new UserManager($pdo);
7 $user = $manager->connect('john@doe.com', 'p4$$w0rd');
8
9 if (!in_array('ROLE_DELIVERABLE', $user->getRoles())) {
10     throw new Exception('Accès interdit');
11 }

```