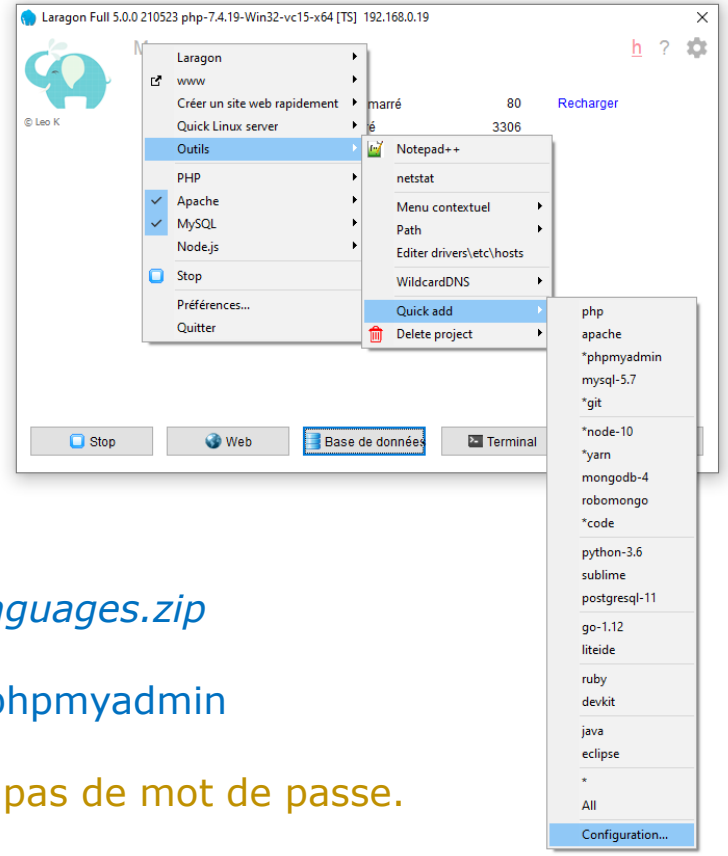
SQL

-Niveau débutant-

Configurations (1/2)

Installation



Nous allons utiliser le Système de Gestion de Bases de Données MySQL.

Pour cela, nous devons tout d'abord installer phpMyAdmin sur Laragon.

Remplacez le lien (ligne 8) de phpmyadmin par :

https://files.phpmyadmin.net/phpMyAdmin/5.1.1/phpMyAdmin-5.1.1-all-languages.zip

Vous pouvez maintenant retourner dans le menu Quick add et cliquer sur *phpmyadmin

L'identifiant de connexion de base est root pour le nom d'utilisateur, et pas de mot de passe.

S'il s'agit de votre 1ère installation, vous aurez certainement une erreur concernant la phrase secrète.

Pour la corriger, rendez-vous dans laragon\etc\apps\phpMyAdmin\config.inc.php et assurez-vous que la variable $cfg['blowfish_secret'] contienne au moins 32 caractères.

Configurations (2/2)

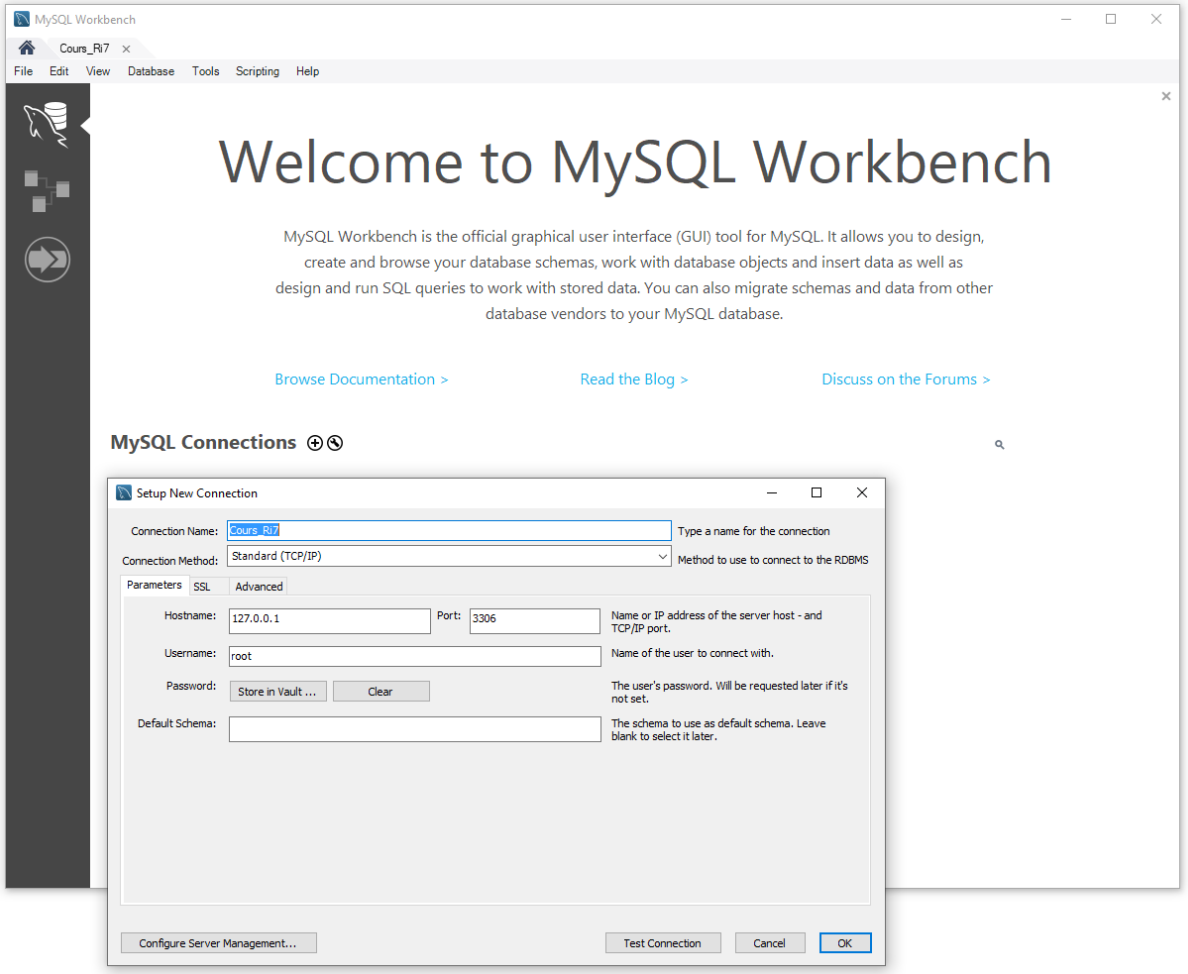
Un dernier logiciel

Afin de nous simplifier la vie dans l'utilisation de mySQL, nous allons utiliser un logiciel tiers à phpMyAdmin : **MySQL Workbench**.

Rendez-vous sur https://www.mysql.com/fr/products/workbench/ puis téléchargez l'application.

Appuyez simplement sur le **+** à côté de MySQL Connections et complétez les champs comme suit.

Nous sommes prêt à nous lancer !



Introduction (1/3)

Définitions

Un peu de théorie avant d'aller plus loin, nous allons voir le lexique basique du SQL.

Base de donnée : Il s'agit d'un système de stockage de données de façon organisée et hiérarchisée. Les données stockées ne sont pas perdues à chaque génération de page et le mot le plus important ici est qu'elles sont organisées, nous permettant de les récupérer très simplement.

SQL : C'est le langage utilisé pour manipuler des bases de données. Il n'est pas comme tous les langages que vous avez pu voir, il possède une syntaxe qui lui est propre permettant de créer des requêtes.

MySQL : C'est un Système de Gestion de Bases de Données (SGBD). Les bases de données sont des systèmes très complexes, c'est le SGBD qui va nous permettre de communiquer avec elles de manière simple. Il en existe plusieurs mais nous allons nous concentrer sur MySQL qui est gratuit et parmi les plus utilisés.

Introduction (2/3)

Définitions

phpMyAdmin: Nous pouvons interagir de 2 manières avec MySQL, soit directement via du code PHP, soit à l'aide d'un logiciel nous proposant une interface dédiée comme phpMyAdmin. Ces 2 interactions dépendent de ce que l'on souhaite faire, pour toute **opération manuelle** (création de base de données, modification, récupération de données) nous utiliserons phpMyAdmin. Pour toute **opération dynamique** (création d'un nouvel utilisateur à son inscription, utilisation de données sur notre page, …) nous utiliserons PHP/MySQL.

Lexique

Précisons un peu tout ça. Celle qui stocke toutes nos données s'appelle la **base de données**. Dans cette base de données, nous pouvons classer des informations par thème/catégorie/utilité dans ce que l'on appelle des **tables**. Ces tables, considérez-les comme des tableaux, contiennent des lignes : des **entrées/tuples**, et des colonnes/attributs.

	ID	Nom	Prenom	Age
	1	VIALE	Brendan	82
	2	ARCHI	Bald	48

Chaque entrée est ici une nouvelle personne, chaque colonne correspond aux informations que l'on souhaite recueillir sur cette personne. L'intersection d'une entrée et d'une colonne s'appelle un **champ**.

Introduction (3/3)

Lexique

BASE DE DONNÉE : SITE E-COMMERCE

TABLE : USER

ID	NOM	PRÉNOM	ADRESSE	MAIL
1	VIALE	BRENDAN	50 Boulevard du Ping Pong	brendan.viale@ping.fr
2	LU	GERTRUDE	10 Rue des Biscuits	gertrude.lu@live.fr

CHAMP

ENTRÉE

TABLE : ORDER

ID	OBJET	QUANTITÉ	PRIX
1	Côton-tige diamant édition premium deluxe	2	300000000
2	Cure-dents	25	50

Création de table (1/4)

Nous y sommes, bienvenue dans ce nouveau langage ! Concrètement, nous allons communiquer avec MySQL grâce à ce que nous appelons des requêtes. Commençons par créer notre table.

Le requête dont on va avoir besoin est **CREATE TABLE**, elle s'accompagne du nom de notre table puis des colonnes entre parenthèses. De la même manière que des variables, nous allons assigner à chaque colonne un type de données, dont voici les plus courants :

- **INT** : nombre entier
- **VARCHAR(*length*)** : texte court, la « length » est facultative mais recommandée pour définir une taille maximale
- **TEXT** : texte long
- **DATE** : date (jour, mois, année)

Ne mettez pas d'espace dans vos noms de colonne. Cela est toléré mais vous obligera à utiliser le caractère « ` » autour du nom lors de requêtes.

Création de table (2/4)

Si nous reprenons mon tableau précédent, pour créer cette table j'aurais pu écrire :

	ID	Nom	Prenom	Age
	1	VIALE	Brendan	82
	2	ARCHI	Bald	48

```
CREATE TABLE user(
    ID INT,
    Nom VARCHAR(30),
    Prenom VARCHAR(30),
    Age INT
);
```

Cela marche, vous pouvez essayer, sauf que c'est incomplet ! Premier point très important, toute table doit posséder une colonne (ou un ensemble de colonnes) dont les valeurs seront uniques pour chaque entrée, nous permettant de pouvoir les dissocier l'une de l'autre. Le plus souvent, nous créons une colonne ID que l'on incrémente de 1 pour chaque nouvelle entrée. Pour définir une colonne comme clé primaire, il faut utiliser l'attribut **PRIMARY KEY**. Pour que notre champ s'incrémente, nous indiquerons également l'attribut **AUTO_INCREMENT**.

```
ID INT AUTO_INCREMENT PRIMARY KEY,
```

Création de table (3/4)

Attributs et contraintes

Nous venons d'en voir 2, mais il existe bien sûr d'autres attributs ou contraintes dont nous allons pouvoir nous servir :

- **NOT NULL** : Le champ ne peut être vide (les null ne sont pas acceptés)
- **UNIQUE(**nom de la colonne**)** : Chaque champ est unique dans sa colonne
- **PRIMARY KEY** : Définit une ou plusieurs colonne(s) comme clé primaire, il ne peut y avoir qu'une seule clé primaire par table
- **FOREIGN KEY** : Nous y reviendrons
- **CHECK** : Définit des conditions pour remplir un champ (il doit être supérieur à, différent de, …)
- **DEFAULT** value : Définit une valeur de base pour un champ s'il est laissé vide
- **AUTO_INCREMENT** : Incrémente de 1 la colonne pour chaque nouvelle entrée
- **UNSIGNED** : Pour les nombres, limite à des nombres positifs (0 inclus)

Création de table (4/4)

Exemple

Voici donc le code final, on a rajouté l'attribut **NOT NULL** pour s'assurer que l'on remplisse bien ces champs.

```
CREATE TABLE user(
    ID INT AUTO_INCREMENT PRIMARY KEY,
    Nom VARCHAR(30) NOT NULL,
    Prenom VARCHAR(30) NOT NULL,
    Age INT NOT NULL
);
```

TD1

Voir ressource TD1

Manipulation des données (1/5)

Insérer

Nos tables sont créées mais elles sont vides pour le moment. Pour rajouter des entrées dans notre table, il faut utiliser l'instruction **INSERT INTO**. Voici sa syntaxe :

```
INSERT INTO film VALUES(
    "Alien",
    1979,
    "Ridley Scott",
    116,
    "Durant le voyage-retour du cargo spatial Nostromo après une mission commerciale de routine,
l'équipage, cinq hommes et deux femmes plongés en hibernation depuis dix mois sont tirés de leur
léthargie plus tôt que prévu par l'ordinateur de bord du vaisseau. Ce dernier a en effet capté des
signaux radio inconnus dans l'espace et, du fait d'une clause attenante à leur contrat de
navigation, l'équipage du vaisseau est tenu de vérifier tout indice de vie extraterrestre.")
```

Vous devez écrire l'instruction INSERT INTO suivie de la table voulue puis du mot-clé VALUES. Vous pouvez ensuite indiquer entre parenthèses chaque champ dans le bon ordre de création des colonnes.

- Vous pouvez spécifier manuellement les colonnes que vous souhaitez remplir en les indiquant après le nom de la table, entre parenthèses.

```
INSERT INTO film(Titre, Année, Réalisateur, Durée, Résumé) VALUES(
```

Manipulation des données (2/5)

Modifier

Si nous souhaitons maintenant modifier des données, nous devons utiliser l'instruction **UPDATE**.

```
UPDATE film SET Année = 1980 WHERE Titre = "Alien"
```

La syntaxe est donc UPDATE *table* SET *colonne = nouvelle valeur* suivi d'une nouvelle instruction, **WHERE**.

Ce mot-clé est très important dans le cas des **UPDATE** (mais aussi dans bien d'autres cas, retenez-le !). Il nous permet d'écrire une condition pour cibler les entrées que la requête doit affecter. Par exemple, dans notre cas, je demande de changer l'année uniquement du film qui a pour titre « Alien ».

Si nous ne l'écrivons pas, dans un UPDATE, le changement s'opérera sur l'ensemble des entrées, donc tous nos films seraient ici créés en 1980.

Manipulation des données (3/5)

Supprimer

L'instruction est cette fois-ci **DELETE FROM**.

```
DELETE FROM film WHERE Titre = "Alien"
```

La syntaxe est donc DELETE FROM *nom de table* WHERE *condition*.

Nous retrouvons encore notre WHERE pour cibler les entrées à supprimer.

Manipulation des données (4/5)

WHERE

Avant d'aller plus loin, voici les opérateurs les plus répandus, et leur syntaxe ci-dessous.

```
Where Année IN (valeur1, valeur2, valeur3)

Where Année BETWEEN 'valeur1' AND 'valeur2'
```

LIKE est un peu particulier, il repose sur le même principe que les expressions régulières, avec seulement 2 symboles à connaître :
« **%** » qui remplace une infinité de caractères
« **_** » qui remplace un seul caractère.

```
Where Année LIKE 'A%'
Where Année LIKE '%A'
Where Année LIKE 'A_B'
```

Il est également possible de combiner plusieurs conditions grâce aux mots-clés **AND** et **OR**.

Opérateur	Description
=	Égale
<>	Pas égale
!=	Pas égale
>	Supérieur à
<	Inférieur à
>=	Supérieur ou égale à
<=	Inférieur ou égale à
IN	Liste de plusieurs valeurs possibles
BETWEEN	Valeur comprise dans un intervalle donnée (utile pour les nombres ou dates)
LIKE	Recherche en spécifiant le début, milieu ou fin d'un mot.
IS NULL	Valeur est nulle
IS NOT NULL	Valeur n'est pas nulle

Manipulation des données (5/5)

Sélectionner

D’accord nous avons créé des tables, nous savons les remplir et les modifier mais comment peut-on les récupérer ? L’instruction à utiliser est **SELECT** sélecteur **FROM** table

```
SELECT * FROM film
```

Le sélecteur peut être le nom d’une colonne, de plusieurs colonnes séparées par une « , » ou encore du symbole « * » qui permet de récupérer toutes les colonnes à la fois.

Essayez donc cette commande, vous verrez que MySQL Workbench vous affiche bien les lignes sélectionnées !

De la même manière que n’importe quelle autre requête, vous pouvez tout à fait combiner SELECT avec des conditions WHERE pour cibler une entrée à récupérer.

TD2

Voir ressource TD2

Manipulation des tables

Ne paniquez pas si vous avez fait une erreur lors de la création de votre table, vous n'avez pas besoin de tout supprimer et recommencer ! Notre sauveur s'appelle **ALTER TABLE**.

Nous allons voir 4 instructions différentes :

- **ADD** : Ajoute une nouvelle colonne

```
ALTER TABLE film ADD ActeurPrincipal VARCHAR(60) NOT NULL
```

- **DROP** : Supprime une colonne

```
ALTER TABLE film DROP ActeurPrincipal
```

- **MODIFY** : Modifie le type d'une colonne. Attention à ne pas changer vers un type non compatible avec le précédent, cela pourrait corrompre votre base de données.

```
ALTER TABLE film MODIFY Année DOUBLE
```

- **CHANGE** : Renomme une table, en indiquant dans l'ordre l'ancien nom et le nouveau. Il faut bien préciser le type de la colonne à la fin.

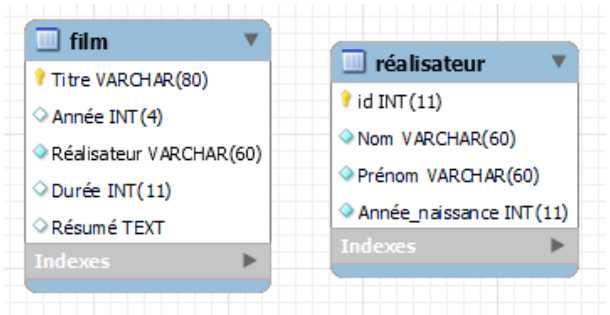
```
ALTER TABLE film CHANGE Durée DuréeMinutes INT(11)
```

TD3

Voir ressource TD3

Clés primaires et étrangères (1/3)

C’est ici que les choses commencent à se compliquer. Vous vous souvenez certainement de la clé primaire que l’on a déjà vue, je vous ai dit qu’il fallait choisir une valeur unique permettant d’identifier une ligne précise dans votre table. Nous pencherons toujours pour le choix le plus logique et simple. Si aucun choix n’est disponible ou s’il est trop complexe, nous allons devoir créer une clé artificielle : une colonne qui ne sert qu’à différencier chaque ligne, le plus souvent un identifiant.

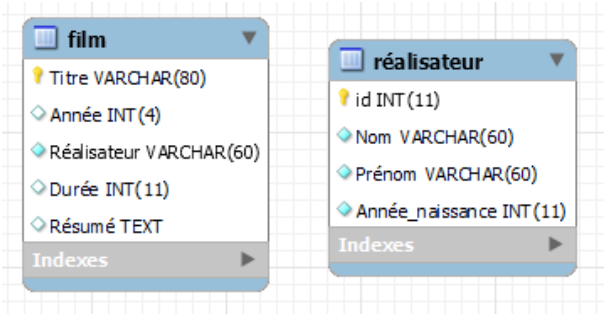


Regardons la table film, quelles sont les différentes possibilités de clés uniques ?

Réfléchissons de la même manière pour le réalisateur.

Clés primaires et étrangères (2/3)

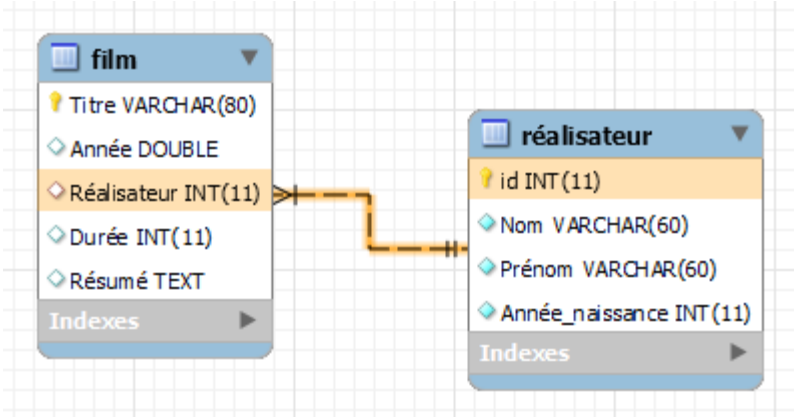
Nous entrons enfin dans les bases de données relationnelles ! Vous n'aurez quasiment jamais une seule table dans votre base de données, et la plupart du temps, vos tables seront liées entre elles. Si nous revenons à notre précédent exemple, notre table **film** contient les réalisateurs, tandis que la table **réalisateur** contient leur nom, prénom et année de naissance. Pour faire le lien entre ces 2 tables, nous allons devoir créer une **clé étrangère**. Cette clé est en fait une colonne de notre table film qui fait référence à la clé primaire de notre table réalisateur.



Clés primaires et étrangères (3/3)

Pour définir une colonne comme clé étrangère, voici la syntaxe :

```
CREATE TABLE film (
    Réalisateur INT,
    FOREIGN KEY (Réalisateur) REFERENCES réalisateur(id)
);
ALTER TABLE film ADD FOREIGN KEY (Réalisateur) REFERENCES réalisateur(id)
```



La 1ère requête correspond à la manière de déclarer une clé étrangère à la création de la table, la seconde est pour l'ajout d'une clé étrangère sur une table déjà créée.

Jointures (1/4)

Maintenant que nous commençons à penser relationnel, attaquons avec les **jointures**.

Toujours d'après le même exemple, je souhaite maintenant récupérer les films avec le nom et le prénom de leur réalisateur. Pour parvenir à cela, nous allons avoir besoin des jointures. Il en existe 2 types :

- **Internes** : Ne renvoie que les entrées qui ont une valeur commune entre les tables voulues
- **Externes** : Renvoie toutes les données, même sans similitude

Par exemple, si nous créons un film en lui omettant son réalisateur, une jointure interne ne l'affichera pas alors que l'externe oui.

Jointures (2/4)

Jointures Internes

Donc, imaginons que dans ma table film je n'ai rempli que 2 films : Alien et Star Wars. J'ai cependant omis d'attribuer un réalisateur à Star Wars. Voyons ma commande INNER JOIN :

```
SELECT f.Titre, f.Année, r.Nom, r.Prénom from film f INNER JOIN réalisateur r ON f.Réalisateur = r.id
```

D'où sortent tous ces f. et ces r. quelque chose ? Il est possible, en SQL, de simplifier le nom d'une table si nous le souhaitons. Pour cela, il suffit de rajouter le diminutif que nous souhaitons après l'appel de la table dans notre requête. Par exemple, il s'agit ici de « film f » et « réalisateur r ». L'intérêt est de ne pas se perdre en longueur lorsque nous manipulons des requêtes sur plusieurs tables, et de ne pas risquer de doublons de nom de colonnes. En indiquant le diminutif suivi d'un « . » et du nom de la colonne voulue, nous pouvons sélectionner les colonnes de notre choix, de la table de notre choix et dans l'ordre souhaité.

Pour en revenir aux jointures, le mot-clé est ici **INNER JOIN** suivi de la table, puis de **ON** et des colonnes qui correspondent entre chaque table. Voici le résultat :

	Titre	Année	Nom	Prénom
▶	Alien	1979	Scott	Ridley

Jointures (3/4)

Jointures Externes

Place maintenant aux jointures externes qui, elles, n'omettent aucune donnée. Les voici :

- **LEFT JOIN** : Renvoie toutes les données de la table de gauche (la 1ère appelée), ainsi que celles de droite qui respectent la condition

```
SELECT f.Titre, f.Année, r.Nom, r.Prénom from film f LEFT JOIN réalisateur r ON f.Réalisateur = r.id
```

	Titre	Année	Nom	Prénom
▶	Alien	1979	Scott	Ridley
	Star Wars 1	1999	NULL	NULL

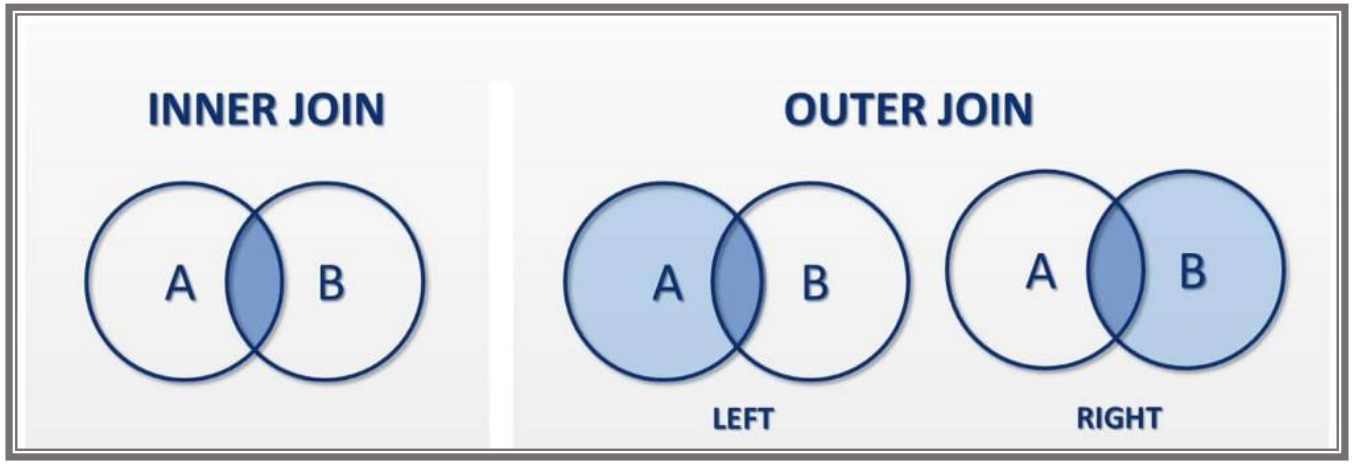
- **RIGHT JOIN** : Renvoie toutes les données de la table de droite, ainsi que celles de gauche qui respectent la condition

```
SELECT f.Titre, f.Année, r.Nom, r.Prénom from film f RIGHT JOIN réalisateur r ON f.Réalisateur = r.id
```

	Titre	Année	Nom	Prénom
▶	Alien	1979	Scott	Ridley

Jointures (4/4)

Voici une image explicative de ce que je viens de vous écrire :



TD4

Voir ressource TD4

Fonctions d'agrégation (1/3)

Le langage SQL nous offre la possibilité d'utiliser des fonctions d'opérations statistiques dans nos requêtes.
Vous allez comprendre :

- **AVG()** : Calcule la moyenne de la colonne
- **COUNT()** : Renvoie le nombre de lignes ciblées par notre requête
- **MAX()** : Renvoie la valeur maximum de la colonne
- **MIN()** : Renvoie la valeur minimum de la colonne
- **SUM()** : Calcule la somme de la colonne

Par exemple, cette requête affichera pour chaque commune uniquement la ligne qui contient leur plus grand indice de fréquentation, dans l'ordre des communes

```
SELECT Commune, MAX(`Indice de fréquentation`) FROM frequentation_cinema GROUP BY Commune
```

Fonctions d’agrégation (2/3)

Le Partitionnement

Lorsque nous utilisons une fonction d’agrégation, nous devons faire appel à l’instruction **GROUP BY** suivie du nom de la colonne, et qui se place après le WHERE. Elle permet d’associer le résultat de notre fonction à une autre colonne.

Essayez donc d’afficher ce tableau :

Nom	Nombre d'années référencées	Superficie
Arles	59	759
Marseille	60	240
Nice	58	72
Paris	0	105

Fonctions d'agrégation (3/3)

HAVING

Pour effectuer une restriction sur une fonction d'agrégation, la clause **WHERE** ne fonctionnera pas. En effet, **WHERE** ne prend en compte que les conditions **avant** que l'agrégation soit effective ! Pour palier ce problème, nous allons utiliser **HAVING** de la même manière.

Par exemple, cette requête affichera chaque commune avec le nombre d'années sur lesquelles elle a été enregistrée dans la table, à condition qu'elle l'ait été au moins 55 fois.

```
SELECT COMMUNE, COUNT(Année) FROM frequentation_cinema GROUP BY COMMUNE HAVING COUNT(Année) > 55
```

HAVING doit être placé en toute fin de requête

Le Tri

Nous pouvons trier le résultat de notre requête selon une (ou plusieurs) colonne de notre choix ! Par exemple, j'aimerais afficher les villes et leur superficie, en triant par ville dans l'ordre alphabétique. Pour cela, nous utilisons l'instruction **ORDER BY** suivie du nom de la colonne. Nous pouvons y ajouter le mot-clé **ASC** pour trier dans l'ordre ascendant (ordre alphabétique) ou **DESC** pour l'ordre descendant.

```
SELECT Nom, Superficie FROM villes ORDER BY Nom DESC
```

	Nom	Superficie
	Paris	105
	Nice	72
	Marseille	240
	Arles	759

TD5

Voir ressource TD5

PHP & SQL (1/9)

Connexion

Nous avons enfin toutes les connaissances nécessaires pour commencer à travailler avec PHP et SQL ensemble.

Nous allons tout de suite rentrer dans le bain et se baser sur l'API la plus courante et vous permettant le plus de portabilité : PDO. Notez que l'utilisation de PDO fait appel à de la Programmation Orientée Objet, ne paniquez pas nous allons voir cette nouvelle syntaxe en parallèle !

```php
$servername = 'localhost'; // Nom du serveur
$username = 'root'; // Login pour se connecter à votre base de données
$password = ''; // Mot de passe pour se connecter à votre base de données

// On essaie de se connecter à la base de donnée
try{
    //Instruction pour se connecter
    $db = new PDO("mysql:host=$servername;dbname=yolo", $username, $password);
    //On définit le mode d'erreur de PDO sur Exception
    $db->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
    echo 'Connexion réussie';
}

// S'il y a une erreur (exception), on rentre dans le catch
catch(PDOException $e){
  echo "Erreur : " . $e->getMessage();
}
```

Voici le code correspondant à la connexion à votre base de donnée, en PHP.

Connexion

Point qui vous est sûrement nouveau, les blocs **try** et **catch**.

Lorsque nous travaillons avec une base de données, la sécurité est primordiale ! Vous ne voulez pas qu'un utilisateur ait accès à votre base de données, c'est évident. Pour gérer les exceptions que pourraient générer votre connexion (mauvais identifiant, mot de passe, …) ou toute requête SQL de manière générale, nous allons mettre de côté nos *if* et utiliser *try catch*, la meilleure solution à ce jour pour gérer des erreurs en orienté objet.

Concrètement, nous allons insérer le code qui peut soulever des exceptions dans le bloc try (ici notre connexion à notre base de données), puis écrire dans le bloc catch le message d'erreur à afficher si une exception a été soulevée. Notez que le bloc catch accepte un paramètre de type Exception, c'est dans celui-ci que se trouve notre exception. Pour l'afficher, nous devons faire appel à sa méthode getMessage().

Déconnexion

Pour fermer la connexion à votre base de données, il suffit de définir votre variable ayant permis la connexion vers votre base de données à NULL.

```php
//Connexion fermée
$db = null;
```

Écrire des requêtes

Cela va certainement faire des heureux, l'utilisation des requêtes SQL en PHP est très simple, le plus dur est en réalité la requête SQL en elle-même, chose que vous maîtrisez sur le bout des doigts ! Pour écrire une requête, vous devez d'abord créer une variable dans laquelle vous définissez votre code SQL, puis vous l'exécutez. Voici la syntaxe :

```
$codeSQL = "CREATE TABLE Joueur (
    id INT AUTO_INCREMENT PRIMARY KEY,
    Pseudo VARCHAR(30) NOT NULL,
    Race ENUM('Elfe','Nain','Halfling','Orc','Humain'),
    idCaracteristique INT NOT NULL,
    Niveau INT UNSIGNED NOT NULL,
    idEquipement INT NOT NULL,
    Argent INT UNSIGNED NOT NULL)";
$db -> exec($codeSQL);
```

Écrire des requêtes - Autocommit

Imaginons que nous souhaitions effectuer plusieurs requêtes à la suite, donc plusieurs variables $codeSQL que l'on exécute. Cela marcherait, bien sûr, mais nous ferions face à un problème : s'il y a une erreur sur certaines requêtes, nous nous retrouverions avec des requêtes qui seraient bien traitées et d'autres non !

Pour éviter cela nous allons utiliser les méthodes **beginTransaction()**, **commit()** et **rollBack().**

Les méthodes beginTransaction() et commit() vont entourer les requêtes que l'on souhaite vérifier.

L'appel de rollBack() dans le catch permet de restaurer la table à sa valeur initiale si une exception est soulevée.

```php
    //On active la vérification
    $db -> beginTransaction();

    //On définit notre requête SQL
    $codeSQL = "CREATE TABLE Joueur (
        id INT AUTO_INCREMENT PRIMARY KEY,
        Pseudo VARCHAR(30) NOT NULL,
        Race ENUM('Elfe','Nain','Halfling','Orc','Humain'),
        idCaracteristique INT NOT NULL,
        Niveau INT UNSIGNED NOT NULL,
        idEquipement INT NOT NULL,
        Argent INT UNSIGNED NOT NULL)";
    //Puis on l'exécute
    $db -> exec($codeSQL);

    //On insère une ligne dans notre table
    $codeSQL2 = "INSERT INTO Joueur(Pseudo, Race, idCaracteristique, Niveau, idEquipement, Argent) VALUES ('Bluelagon','Elfe',3,10,1,10000000000)";
    $db -> exec($codeSQL);

    //On termine notre vérification
    $db -> commit();

}

// S'il y a une erreur (exception), on rentre dans le catch
catch(PDOException $e){
    //Si une exception a été soulevée dans une requête, on remet les tables à leur état initial
    $db -> rollBack();
    echo "Erreur : " . $e->getMessage();
}
```

Écrire des requêtes – Les requêtes préparées

Si nous souhaitons écrire plusieurs fois une même requête, ou encore pour s'assurer une sécurité accrue sur notre base de données dans le cas d'un remplissage par un utilisateur, nous allons utiliser des requêtes préparées.

Plusieurs syntaxes sont possibles, je vous conseille celle-ci-après.

Relisons tout cela pas à pas :
Nous attribuons à une nouvelle variable $codeSQL la préparation de la requête. Celle-ci se récupère grâce à la méthode prepare() de notre variable PDO. La préparation de la requête est en fait l'écriture de celle-ci, en permettant de définir des valeurs que nous pouvons modifier à notre guise dans notre code. Ces valeurs modifiables sont reconnaissable par leur « : » précédant le nom que nous leur attribuons.

```php
//On prépare notre requête
$codeSQL = $db->prepare(
    "INSERT INTO réalisateur(Nom,Prénom,Année)
    VALUES (:nom, :prénom, :année)"

//On attribue des variables à nos paramètres
$codeSQL -> bindParam(':nom',$nom);
$codeSQL -> bindParam(':prénom',$prenom);
$codeSQL -> bindParam(':année',$annee);

//On définit les 1ères valeurs que nous souhaitons entrer
$nom="Scott";
$prenom="Ridley";
$annee=1937;
//Puis on exécute la requête
$codeSQL -> execute();

//On définit les valeurs de la seconde ligne que nous souhaitons entrer
$nom="Scott";
$prenom="Ridley";
$annee=1937;
//Puis on ré-exécute la requête
$codeSQL -> execute();
```

Écrire des requêtes – Les requêtes préparées

Pour modifier ces valeurs, nous allons faire appel à la méthode bindParam, qui accepte 5 paramètres dont 2 obligatoires, que j'utilise ici.
Nous utilisons donc cette méthode sur notre variable codeSQL (qui contient notre requête) et on indique dans un premier temps le champ que nous souhaitons modifier, puis la valeur que nous lui attribuons. Notez que cette valeur est en réalité une référence, la véritable valeur sera donnée lors de l'exécution de la requête.
Enfin, nous faisons appel à la méthode execute() pour exécuter notre requête. Il est possible de le faire autant de fois que nous le souhaitons, chaque exécution prendra en compte les dernières valeurs stockées dans nos variables, regardez-donc ma 2ème exécution pour comprendre !

```php
//On prépare notre requête
$codeSQL = $db->prepare(
    "INSERT INTO réalisateur(Nom,Prénom,Année)
    VALUES (:nom, :prénom, :année)"

//On attribue des variables à nos paramètres
$codeSQL -> bindParam(':nom',$nom);
$codeSQL -> bindParam(':prénom',$prenom);
$codeSQL -> bindParam(':année',$annee);

//On définit les 1ères valeurs que nous souhaitons entrer
$nom="Scott";
$prenom="Ridley";
$annee=1937;
//Puis on exécute la requête
$codeSQL -> execute();

//On définit les valeurs de la seconde ligne que nous souhaitons entrer
$nom="Scott";
$prenom="Ridley";
$annee=1937;
//Puis on ré-exécute la requête
$codeSQL -> execute();
```

Écrire des requêtes – Les requêtes préparées

Nous pourrions utiliser d'autres syntaxes, par exemple nous pouvons omettre les bindParam() pour intégrer directement nos variables à l'exécution :

```php
$nom="Scott";
$prenom="Ridley";
$annee=1937;
//On exécute la requête
$codeSQL -> execute(array(
    ':nom' => $nom,
    ':prenom' => $prenom,
    ':annee' => $annee
));
```

Pour les curieux, il existe également bindValue() qui fonctionne de la même manière que bindParam(). La seule différence est que bindValue() associe la valeur de la variable au moment où on l'appelle, tandis que bindParam() ne le fait qu'à l'exécution de la requête. Dans ce dernier, si nous changeons les valeurs des variables entre notre bind et notre execute(), cela sera bien pris en compte, alors qu'avec bindValue() non.

PHP & SQL (9/9)

Récupérer nos données

Il ne me reste plus qu'à vous montrer quelques fonctions utiles dans l'utilisation de vos données en PHP :

- La méthode **rowCount()** nous retourne le nombre de lignes affectées par notre requête.

```php
$codeSQL = $db->prepare("SELECT * FROM réalisateur");
$codeSQL -> execute();
//On récupère le nombre de lignes retournées
$nbLignes = $codeSQL->rowCount();
```

- Certainement la plus importante, **fetchAll()** va vous permettre de convertir le résultat de votre requête en un tableau. Vous pourrez donc récupérez librement les données pour les utiliser dans votre code ! Le paramètre PDO::FETCH_ASSOC va nous permettre de créer un tableau associatif comprenant comme clé le nom de la colonne et comme valeur le champ correspondant. Essayez donc un print_r($tableauRequete) !

```php
$codeSQL = $db->prepare("SELECT * FROM réalisateur");
$codeSQL -> execute();
//On récupère le résultat de notre requête sous forme de tableau associatif
$tableauRequete = $codeSQL->fetchAll(PDO::FETCH_ASSOC);
```

PROJET

Voir ressource Projet