SQL

LE SQL

SQL (**S**tructured **Q**uery **L**anguage) est un langage permettant d'interroger des bases de données relationnelles.
Il permet principalement de rechercher, ajouter, modifier ou supprimer des données.

Qu'est-ce qu'une base de données ?
C'est un ensemble d'informations organisé de manière à être facilement accessible.
Souvent les données sont organisées dans des tables regroupant les informations de la donnée qu'elles contiennent.

HISTORIQUE

SQL a été créé en 1974, par **IBM**, puis normalisé en 1986, il s'agit du langage le plus reconnu par les système de gestion de base de données (SGBD, DBMS en anglais DataBase Management System).

Il est maintenant détenu par **Oracle.**

SGBD ?

Un SGBD est un programme qui gère :

- Le stockage des données (fichiers)
- Gère l'accès aux données (utilisateurs, requêtes)
- Gère la sécurité (paramètres d'accès, filtres d'IP, etc)
- Gère les performances (paramètrables)

Il en existe plusieurs, les plus utiliser sont :

- MySQL
- MariaDB

SGBD(R) ?

On parle souvent de SGBD relationnel, mais ce qui est corresponds à quoi ?

C'est un moyen de stocker les données de manières « structurées » afin d'en faciliter l'accès et l'utilisation, on appelle ça des « **tables** ».

SGBD(R) : LES TABLES

Une table représente un objet, une entité de la vie courante, que l'on souhaite stocker dans une base de données, une base de données peut avoir plusieurs tables.

Une table est composé de « **colonnes** », représentant un descriptif de l'objet, comme le « **nom** » pour un utilisateur, ou son « **email** ».

Elle est aussi composé de lignes, où seront sauvegarder la valeur de la donnée.

Utilisateur

nom	email	
Dubois	j.dubois@gmail.com	⇦ ligne 2
Toto	b.toto@gmail.com	⇦ ligne 1

⇧ colonnes ⇧

SGBD(R) : LES TABLES : ID

Chaque table doit avoir un **identifiant unique,** il a pour but de déclarer une colonne de la table qui sera une, c'est à dire que sa valeur soit différente pour chaque ligne de la table, permettant d'avoir une représentation unique pour chaque ligne.
On pourra s'en servir par la suite pour relier les tables en elles et ainsi faire passer des informations d'une table à d'autres.

Utilisateur

id	nom	email	
1	Dubois	j.dubois@gmail.com	⇦ ligne 2
2	Toto	b.toto@gmail.com	⇦ ligne 1

⇧ colonnes ⇧

INSTALLATION DE MYSQL

Souvent un serveur MySQL est inclus lorsque l'on prend un hébergement web.
Heureusement il est possible d'en avoir un pour tout ce qui est développement local.

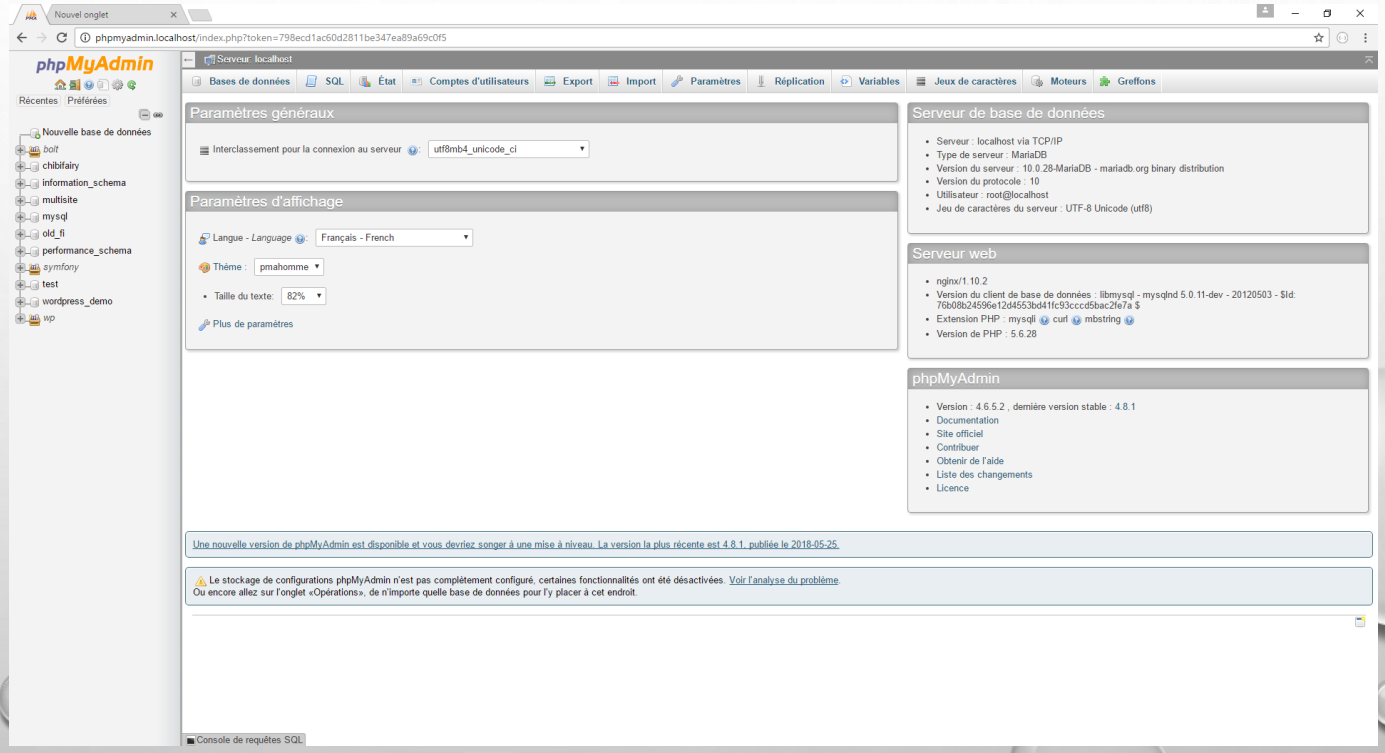
	Windows	Linux	MacOS	HTTP server	SGBDR
WAMP	OUI	non	non	Apache	MySQL & MariaDB
MAMP	OUI	non	OUI	Apache & Nginx	MySQL
XAMPP	OUI	OUI	OUI	Apache	MariaDB

WAMP SERVER

Pour la suite nous utiliserons Wamp serveur, car il permet d'utiliser aussi bien MySQL que MariaDB.

https://www.wampserver.com/

Wamp serveur permet d'avoir une interface graphique permettant la gestion d'un serveur MySQL en local.



CRUD

Le CRUD représente 4 opérations principales utilisées dans un système :

- **C**reate : création/ajout de données
- **R**ead : lecture/affichage de données
- **U**pdate : modification de données
- **D**elete : suppression de données

Chacune de ces opérations peut être effectuée sur une base de données MySQL et possède donc sa propre instruction pour que l'opération soit effectuée.

READ : SELECT

Afin de récupérer ou afficher des données depuis une base de données, il faut utiliser l'opération **SELECT** :

```
SELECT expression FROM nomTable;
```

Lors d'un select il faut toujours préciser depuis quelle table on souhaite afficher les données, avec le mot-clé « FROM ». L'expression demandée peut prendre plusieurs valeurs :

```
SELECT * FROM students;
```

« * » représente le fait de vouloir afficher toutes les colonnes de la table

```
SELECT email FROM students;
```

« email » est la colonne spécifique que l'on veut afficher

```
SELECT last_name, first_name FROM students;
```

« last_name » et « first_name » sont les colonnes spécifiques que l'on veut afficher

READ : ALIAS

Il est possible de renommer le nom des colonnes affichées en utilisant le mot-clé alias « **AS** » :

```
SELECT expression AS nomAlias FROM nomTable;
```

SELECT - FONCTIONS

Il est possible de préciser différentes fonctions de base de SQL lors d'un SELECT :

```
SELECT COUNT(*) FROM students;
```

« count » permet de compter le nombre de lignes présentes dans une table

Vous pouvez consulter les autres fonctions existantes :

https://sql.sh/fonctions

SELECT - IF

Il est possible de préciser différentes fonctions de base de SQL lors d'un SELECT :

```
SELECT last_name, IF(height >= 180, 'grand', 'petit') FROM students;
```

Ici le « IF » permet de vérifier si la valeur de la colonne « height » est supérieure ou égale à 180, alors on affichera « grand » sinon « petit »

SELECT - WHERE

Avec un « SELECT » il est possible de mettre en place des conditions, afin de filtrer les informations à afficher, notamment le « WHERE » :

```
SELECT expression FROM table WHERE nomColonne OPERATOR value;
```

« OPERATOR » prends les valeurs des différents opérateurs de conditions, comme « **<=** » ou « **>=** » ou « **!=** » ou « **=** », « **<** » etc.

```
SELECT * FROM students WHERE gender = 'F';
```

Filtre les étudiants selon la colonne « **gender** » de valeur « **F** »

Au sein d'un « WHERE », il est possible d'ajouter des conditions « OR » ou « AND » afin de cumuler les filtres.

SELECT - WHERE

Lors d'un test sur une chaîne de caractères, ou des dates, il est obligatoire de mettre des apostrophes simples autour, comme pour ce test :

```
SELECT * FROM students WHERE gender = 'F';
```

Alors que pour des entiers, il n'est pas nécessaire :

```
SELECT * FROM students WHERE height >= 170;
```

Pour les chaînes de caractères, on peut aussi utiliser le mot-clé « **LIKE** », qui cumuler à des « % » peut vérifier qu'une chaîne de caractères contient tel ou tel caractère :

```
SELECT * FROM students WHERE last_name LIKE '%co%';
```

Ici on veut les étudiants dont le nom contient n'importe quels caractères, puis « co » puis n'importe quels caractères

SELECT - LIMIT

« LIMIT » permet de limiter le nombre de résultat en sortie, au lieu de tous les avoir.

```
SELECT expression FROM table LIMIT offset, quantity;
```

- quantity : le nombre de donnée à faire ressortir
- offset : d'où commencer à afficher les donner

Il est possible d'omettre l'offset et ainsi sa valeur sera équivalente à 0

```
SELECT * FROM students LIMIT 5, 10;
```

Affiche 10 étudiants à partir du 5°

```
SELECT * FROM students LIMIT 5;
```

Affiche 5 étudiants

SELECT – ORDER BY

En complétant notre opération « SELECT » avec un « ORDER BY » on peut trier les données que l'on récupère :

```
SELECT expression FROM table ORDER BY nomColonne DIRECTION;
```

« DIRECTION » prends soit la valeur « ASC » (Croissant) ou « DESC » (Décroissant).
La direction est facultative, et par défaut sa valeur est « ASC ».

```
SELECT * FROM students ORDER BY last_name ASC;
```

Trie les étudiants par « nom » croissant, du plus petit au plus grand (ordre alphabétique)

SELECT – GROUP BY

L'instruction « GROUP BY » permet de regrouper les données récupérées selon un critère.

```
SELECT expression FROM table GROUP BY nomColonne;
```

```
SELECT * FROM students GROUP BY gender;
```

Affiche les étudiants regroupés par « gender »

SELECT – HAVING

Le mot clé « **HAVING** » est utilisé comme un « **WHERE** » sauf qu'il permet de tester le résultat de fonction, que le « **WHERE** » ne peut pas faire.

```
SELECT expression FROM table HAVING fonction OPERATEUR test;
```

```
SELECT COUNT(*), gender FROM students GROUP BY gender HAVING COUNT(*) > 30;
```

Souvent le « HAVING » va fonctionner de paire avec le « GROUP BY » pour tester la donnée qui a été regroupé.
C'est-à-dire que cet exemple on va compter les étudiants H, F et afficher ceux étant supérieur à 30.

SELECT ORDRE DES OPERATIONS

On a vu plusieurs opérations possibles dans une requête SQL, mais elles ont toutes une position au sein de celle-ci très précise :

```
SELECT expression
FROM table
WHERE condition
GROUP BY nomColonne
HAVING condition
ORDER BY nomColonne
LIMIT offset, quantity
```

REQUÊTES DE JOINTURES

Si l'on fait un « SELECT * FROM students », le souci est que l'on va simplement voir l'id de la classe liée à l'étudiant. C'est un peu dommage, surtout si le but serait d'afficher dans quelle classe il est.
Pour palier à ce problème, il existe en SQL des **jointures** permettant de lier 2 tables entre elle, au moment du select, et ainsi récupérer les colonnes des deux tables.

Il existe plusieurs jointures, mais les plus importantes sont :

- **(INNER) JOIN :** récupère les informations des deux tables liées, tant que la condition de liaison est valide.
- **LEFT JOIN :** récupère toujours les informations de la table de gauche et les informations de la table de droite, même si la condition de liaison n'est pas valide.

On parle de « table de gauche » pour la table désignée après le « FROM » et de table de droite pour la table liée par le JOIN

PRIMARY KEY

Une « PRIMARY KEY », ou **clé primaire**, est l'identifiant de la table, c'est-à-dire que c'est la colonne représentant l'identifiant unique de la table.
Souvent lorsque l'on déclare une clé primaire, on ajoute « AUTO_INCREMENT », cela indique à notre SGBD qu'il va devoir gérer sa valeur.
Il faut ensuite bien penser à déclarer la colonne « id » comme PRIMARY KEY, cela se fait après la déclaration des colonnes.

```
CREATE TABLE students (
        id INT NOT NULL AUTO_INCREMENT,
        PRIMARY KEY(id)
);
```

RELATIONS

On pourrait vouloir que deux tables aient une relation entre elles, l'objectif est de pouvoir dire qu'une table a une ligne qu'elle souhaite partager avec des lignes présentes dans d'autres tables, et ainsi éviter de répéter une information.

Par exemple, ici je souhaiterai indiquer qu'un étudiant soit dans une classe, mais une classe peut avoir plusieurs étudiants à l'intérieur.
Une solution serait d'ajouter la « class » dans la table étudiant, mais le soucis est que la classe serait identique et commune à plusieurs étudiants.

Imaginons ensuite que l'on voudrait stocker des informations complémentaires à la classe, par exemple le type de diplôme préparé, la date de début de la classe, l'information serait à répéter pour chaque ligne d'étudiants étant dans la même classe : c'est lourd.

Students
first_name last_name email class

FOREIGN KEY

Le principe de la FOREIGN KEY (ou « clé étrangère ») dépends de la PRIMARY KEY, en effet une fois que l'on a déclaré une PRIMARY KEY, on peut la réutiliser pour indiquer qu'une table connaîtra la PRIMARY KEY d'une autre table, c'est le principe de la FOREIGN KEY.

```
CREATE TABLE class (
        id INT NOT NULL AUTO_INCREMENT,
        PRIMARY KEY(id)
);
```

Toujours sur le même principe de la PRIMARY KEY, on va devoir déclarer une nouvelle colonne, par convention on met souvent le nom de l'autre table qui sera utilisée, suffixée par « **_id** ».
Et enfin il faut déclarer cette colonne comme clé étrangère avec cette syntaxe : « FOREIGN KEY (nomColonne) REFERENCES tableReferencee(colonneReferencee) »

```
CREATE TABLE student (
        id INT NOT NULL AUTO_INCREMENT,
        class_id INT NOT NULL,
        PRIMARY KEY(id),
        FOREIGN KEY (class_id) REFERENCES class(id)
);
```

PRIMARY KEY ET FOREIGN KEY

Voilà le rendu en terme de table une fois cette création de table réalisée :

Students
id
first_name
last_name
email
class_id

Class
id
name

Mais en quoi cette schématisation va résoudre le problème de duplication ?

L'idée est que maintenant la table « students » dépendra de la table « class », via la clé étrangère.

Cette clé étrangère représente une ligne de données, mais de la table « class » cette fois, cela revient à indiquer que pour une ligne de la table « students », j'ai les valeurs d'une ligne de la table « class ».

REQUÊTES DE JOINTURES

Comment fonctionne les jointures ?

Le JOIN choisit se place tout de suite après le « **SELECT** » mais avant le « **WHERE** », la condition pour lier les deux tables se fait très souvent entre la colonne déclarée comme clé étrangère et la colonne de la table qu'elle référence (voir création de la table : FOREIGN KEY) :

```
SELECT colonnes FROM tableGauche
JOIN tableDroite
ON tableGauche.key = tableDroite.key;
```

La condition se place dans le « **ON** », situé après le « **JOIN** ».

Pour l'exemple des étudiants avec les classe, on devrait faire ce « **JOIN** »

```
SELECT * FROM students
JOIN class
ON students.class_id = class.id;
```

LE IN

Il existe un moyen d'effectuer plusieurs tests sur une même table le « IN » :

```
SELECT last_name FROM students WHERE id = 1000 OR id = 1001;
SELECT last_name FROM students WHERE id IN (1000,1001);
```

Le « IN » peut prendre plusieurs valeurs entre parenthèses et si la colonne avant le « IN » a une valeur présente dans le « IN » alors la ligne ressortira dans la recherche.

LES SOUS-REQUÊTES

Le principe des sous-requêtes est de pouvoir effectuer une opération SQL, select, insert, update ou delete à partir du résultat d'un autre select.

- Le « SELECT AVG(height) FROM students » calcule la moyenne de la taille de la table student
- Le 1er select permet de récupérer les étudiants dont la taille est supérieure à la moyenne de la table student

```
SELECT *
FROM students
WHERE height > (
    SELECT AVG(height)
    FROM students
);
```

CRÉER UNE BASE DE DONNÉES

Pour créer une base de données dans notre serveur MySQL, on utilise la commande « CREATE DATABASE » :

```
CREATE DATABASE nomBaseDeDonnees
DEFAULT CHARACTER SET utf8;
```

Il est recommandé d'utiliser « UTF-8 » comme encodage de caractères car il possède la plus vaste quantité de caractères.

CRÉER UNE TABLE

Pour créer une base de données dans notre serveur MySQL, on utilise la commande « CREATE TABLE » :

```
CREATE TABLE nomTable (
        colonnes
)
ENGINE = InnoDB
DEFAULT CHARACTER SET = utf8;
```

- « ENGINE » représente un module utilisé par la base de donnée, qui est utilisé pour la création, la lecture et la mise à jour des données de la base. On recommandera « **InnoDB** », qui est le moteur d'enregistrement par défaut de MySQL, car il gère les transactions (exécution de bloc d'instruction sécurisé). Accessoirement, il est le seul à gérer les références de clés étrangères entre les tables.
- Le default character set est par défaut celui de la base de données, auquel cas on aurai pas à le répéter

CRÉER UNE COLONNE : LES TYPES

Afin de créer une colonne dans notre table, on doit lui préciser son type de données à sauvegarder, voici les différents types utilisables :

Types	
Numeric	INT, FLOAT, DECIMAL, BOOLEAN…
String	CHAR, VARCHAR, TEXT, ENUM, …
Date	DATE, DATETIME, TIME, …

CRÉER UNE COLONNE

Lorsque l'on créé une colonne dans une table, on doit préciser son nom, quel type de données sera stockée par celle-ci et si elle est nullable, c'est-à-dire qu'elle peut n'avoir aucune valeur.

```
CREATE TABLE nomTable (
        nomColonne typeColonne nullable
)
ENGINE = InnoDB
DEFAULT CHARACTER SET = utf8;
```

« first_name » est le nom de la colonne, varchar(120) est le type, 120 représente le nombre maximum de caractères et « default null » indique que par défaut la valeur de cette colonne sera vide.

```
CREATE TABLE students (
        first_name VARCHAR(120) DEFAULT NULL
)
ENGINE = InnoDB
DEFAULT CHARACTER SET = utf8;
```

MODIFIER UNE TABLE

Afin de modifier une table on utilise l'opération « ALTER TABLE », et en fonction des actions souhaités, on peut utiliser « ADD COLUMN » ou « DROP COLUMN » :

```
ALTER TABLE newTable
ADD COLUMN  field_2 VARCHAR(255) NOT NULL AFTER field_1;
```

Il suffit de préciser les informations de la colonne, comme dans un « CREATE », l'opération « AFTER » est facultative, par défaut il ajoute la colonne après la dernière.

CREATE : INSERT

Afin d'ajouter des données dans une table d'une base de données, on utilise l'opération « **INSERT… INTO… VALUES** », à chaque fois une nouvelle ligne est ajoutée dans la table.

```
INSERT INTO nomTable (colonne1, colonne2)
VALUES (valeur1, valeur2);
```

```
INSERT INTO students (last_name)
VALUES ('MARTINEZ');
```

Ajoute une ligne dans la table « students », avec seulement le nom « last_name » de renseigner

```
INSERT INTO students
VALUES ('MARTINEZ', 'Paulo', 180, 'M');
```

Si aucune colonne spécifique n'est précisée, alors il faudra toutes les renseigner, tel que spécifier dans la table

UPDATE : UPDATE

Afin de modifier des données dans une base de données, on va utiliser le mot-clé « UPDATE » :

```
UPDATE table
SET colonne1 = valeur, colonne2 = valeur
WHERE colonne OPERATOR value;
```

```
UPDATE students
SET first_name = 'Charlie'
WHERE birth_date = '2000-09-23 12:48:36';
```

Ici on va modifier la colonne « first_name » de la table « students », mais seulement là où la colonne « birth_date » vaut une certaine valeur.

DELETE : DELETE

Afin de supprimer des données dans une base de données, on va utiliser le mot-clé « DELETE FROM » :

```
DELETE FROM table
WHERE condition;
```

```
DELETE FROM students
WHERE height < 140;
```

Ici on va supprimer toutes les lignes de la table « students », où la colonne « height » est inférieure à 140.

VIDER UNE TABLE

Afin de vider une table, on utilise l'opération « TRUNCATE TABLE » :

```
TRUNCATE TABLE nomTable;
```

```
TRUNCATE TABLE students;
```

Ici on va vider la table « students », la table existera toujours dans la base de données, mais elle sera vide. (Truncate réinitialise la valeur de l'auto-increment)

TRANSACTION

Une transaction est un ensemble de requêtes qui seront exécutées d'un seul bloc, mais on choisit le moment où elles prendront réellement effet dans la base de données. Ainsi si jamais l'une d'entre elles ne fonctionnent pas, on peut annuler toutes les requêtes réalisées auparavant, ou simplement valider les requêtes sur la base de données.

Afin de réaliser des transactions il est important de faire un « **SET AUTOCOMMIT = 0** », car en effet par défaut le SGBD est en AUTOCOMMIT = 1, autrement dit dès que l'on exécute une requête elle prend effet immédiatement.
Avec le AUTOCOMMIT = 0, il faut « **COMMIT** » nos requêtes pour qu'elles prennent effet ou les « **ROLLBACK** ».
Afin de déclarer des requêtes qui attendent un commit ou un rollback, on utilise « **START TRANSACTION** »

DECLARATION DE VARIABLES

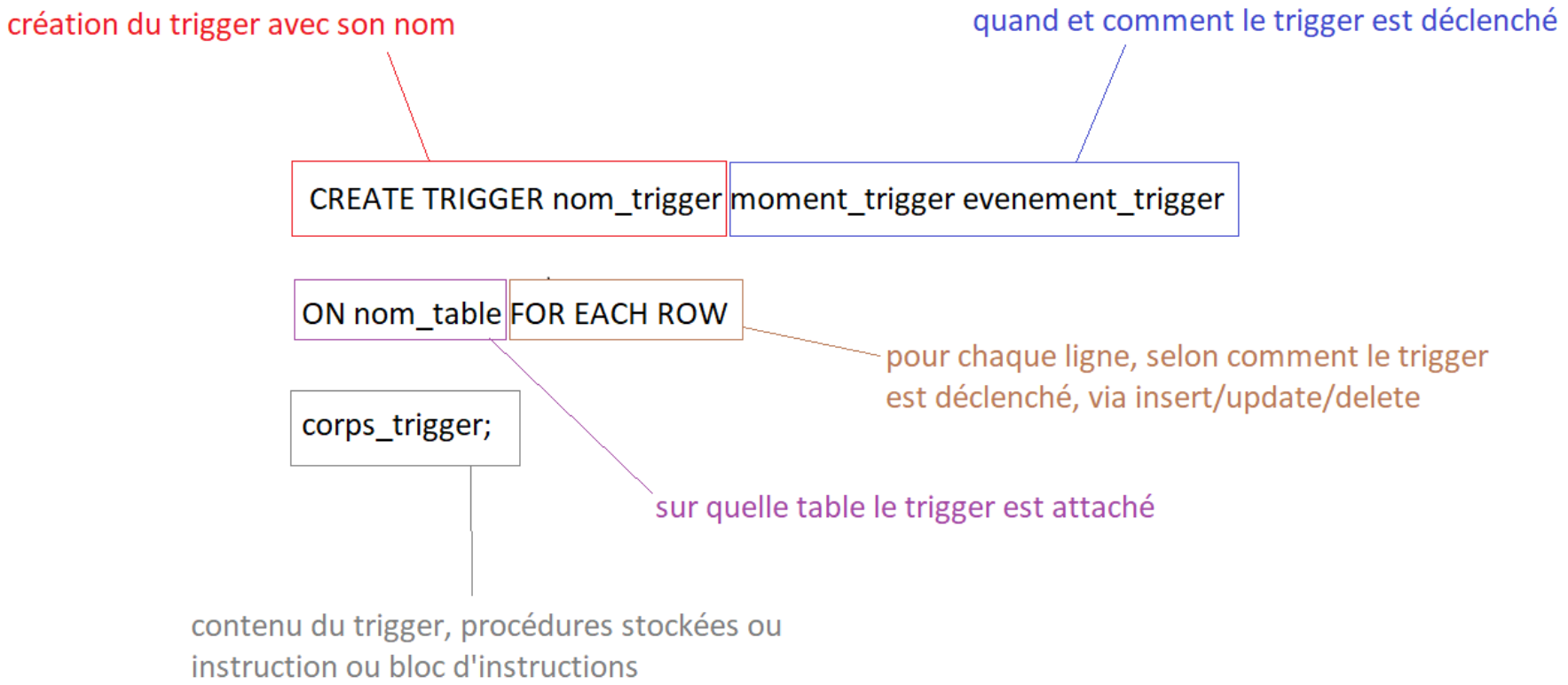
Afin de déclarer une variable en SQL, on utilise « **SET** » et « **@** » devant le nom de la variable :

```
SET @id = (
        SELECT id
        FROM students
        WHERE students.email = toto@gmail.com'
);
```

Dorénavant lorsque l'on utilisera « **@id** » dans nos requêtes suivantes, on aura l'id du student dont l'email vaut « toto@gmail.com »

TRIGGER

Un trigger, ou déclencheur, est un moyen de déclarer du code qui sera exécuté directement lorsque les conditions de déclenchement du trigger seront remplis.



TRIGGER

Un trigger ne peut avoir qu'une seule combinaison de « moment_trigger » et « evenement_trigger » par table, autrement dit une table aura maximum 6 triggers.

- « evenement_trigger » corresponds à : **INSERT**, **UPDATE** ou **DELETE**
- « moment_trigger » corresponds à **BEFORE** ou **AFTER**

Un trigger permet aussi l'utilisation des mots clés : « **OLD** » et « **NEW** », à l'intérieur du « corps_trigger »
« OLD » corresponds à l'ancienne valeur, avant que le trigger ne s'applique et « NEW » à la nouvelle valeur, après que le trigger se déclenche.

Une fois qu'un trigger est mis en place dans la base de données, il se déclenchera tant que les conditions seront remplies ou qu'il soit supprimé, par la commande « **DROP TRIGGER** nom_trigger »

TRIGGER : EXEMPLE


```
DELIMITER |
CREATE TRIGGER listing_before_update BEFORE UPDATE
ON listing FOR EACH ROW
BEGIN
      SET NEW.updated_at = NOW();
END |
DELIMITER ;
```

« DELIMITER » permet d'ajouter un nouveau type de délimiteur dans la portion de code à suivre, autrement dit il exécutera le code jusqu'à rencontrer ce délimiteur.
On s'en sert afin de pallier à une erreur lors de la création de trigger.

REQUÊTE DYNAMIQUE

Cela permet d'avoir un squelette de requête avec un paramètre, ce qui permet de dynamiser les requêtes sans avoir à tout réécrire.

On utilise « **PREPARE nom_request FROM** » puis entre simple quote le select à effectuer, avec un « ? » pour l'endroit où l'on souhaite avoir un paramètre.
Pour la réutiliser on utilise « **EXECUTE nom_request** » et « **USING valeur_param** » :

```
PREPARE select_model_from_brand
FROM '
        SELECT *
        FROM model
        JOIN brand ON model.brand_id = brand.id
        WHERE brand.name = ?
';
```

```
EXECUTE select_model_from_brand USING 'Ferrari';
```

PROCEDURE

Une procédure est comme un fonction qui est stocké sur le serveur MySQL, et que l'on peut exécuter en l'appelant.

Toujours sur le même principe que les trigger, on doit utiliser un « DELIMITER » afin d'éviter des erreurs.
L'opération « **DROP PROCEDURE** » permets d'éviter d'avoir une erreur si la procédure existe déjà.
« **CREATE PROCEDURE nomProcedure** » créer la procédure avec un nom donné, **nomProcedure.** Il faut donner un nom au paramètre de la fonction, ici **idBrand** de type **INT.**
A l'intérieur du **BEGIN … END**, se trouve le code à exécuter

```
DELIMITER //
DROP PROCEDURE IF EXISTS getCa //
CREATE PROCEDURE getCa(idBrand INT)
BEGIN
	SELECT SUM(listing.price), brand.name
	FROM listing
	JOIN model ON model.id = listing.model_id
	JOIN brand ON brand.id = model.brand_id
	WHERE brand.id = idBrand;
END //
DELIMITER ;
```

PROCEDURE

On rappelle une procédure avec le mot-clé « **CALL** » suivi du nom de la procédure.

```
CALL getCa ( 1 );
```

Il est important de noter qu'une procédure ne peux pas renvoyer de valeur, mais peut interpréter les différentes opérations SQL, comme SELECT, INSERT, UPDATE ou DELETE.