

# Projet Pedago Hotel

---

## *Support de cours*

## Les Triggers (Déclencheurs)

### Définition

Un trigger est un bloc de code SQL qui s'exécute automatiquement lorsqu'un événement spécifique (INSERT, UPDATE ou DELETE) survient sur une table.

Le but des triggers est d'intégrer de la logique métier directement au niveau de la base de données. Ils peuvent être mis en place par des administrateurs système ou des DBA (Database Administrators), potentiellement issus d'un service différent de celui des développeurs.

Syntaxe (MySQL) :

```
CREATE TRIGGER nom_trigger
[BEFORE | AFTER] [INSERT | UPDATE | DELETE]
ON nom_table
FOR EACH ROW
BEGIN
    -- instructions SQL
END;
```

Les syntaxes peuvent différencier selon le système de base de données relationnel que vous utilisez (Oracle, PostgreSQL etc...)

Exemple :

```
CREATE TRIGGER maj_date_modification
BEFORE UPDATE ON utilisateurs
FOR EACH ROW
BEGIN
    SET NEW.date_modif = NOW();
END;
```

## Avec des délimiteurs

```
DELIMITER //
CREATE TRIGGER `maj_date_modification` BEFORE UPDATE ON `utilisateurs`
FOR EACH ROW BEGIN
    SET NEW.date_modification = NOW();
END;
//
DELIMITER;
```

Par défaut, **MySQL** utilise **;** pour savoir qu'une commande est terminée.

Mais quand on écrit une trigger, ou plutôt un bloc de code, on peut en avoir plusieurs **;** à l'intérieur du bloc.

```
CREATE FUNCTION ma_fonction()
RETURNS INT
BEGIN
    DECLARE x INT;
    SET x = 10;
    RETURN x;
END;
```

Les 3 dernières lignes ne sont pas lues.

Les triggers sont des composants SQL qui peuvent être lus, modifiés, exportés et supprimés.

Les triggers sont enregistrés dans la table `information_schema`. Pour les lister :

```
SELECT * FROM information_schema.triggers
```

Pour supprimer un trigger :

```
DROP TRIGGER nomDuTrigger;
```

## Mot clé à connaître :

BEFORE	avant l'action.
Ce terme fait référence à l'état des données avant que l'action spécifique ne soit exécutée. Dans le cadre d'un déclencheur (trigger) dans une base de données, la clause BEFORE permet d'observer les valeurs actuelles d'un enregistrement avant que celles-ci soient modifiées.	
AFTER	après l'action.
Le terme AFTER est utilisé pour désigner l'état des données après que l'action a été effectuée. Dans le contexte des déclencheurs, une action AFTER permet d'agir après que les modifications ont été confirmées dans la base de données.	
NEW	pour accéder aux nouvelles valeurs (INSERT ou UPDATE).
La référence NEW est utilisée pour désigner les valeurs qui vont être insérées ou celles qui ont été mises à jour. NEW n'est pas accessible sur un DELETE	
OLD	pour accéder aux anciennes valeurs (UPDATE ou DELETE).
La référence OLD est utilisée pour désigner les valeurs qui ont été insérées ou celles qui ont été mises à jour. OLD n'est pas accessible pour un trigger sur un INSERT.	

## Trigger d'historisation

Une pratique courante est d'enregistrer les anciens enregistrements dans une table historique.

La table historique contient les mêmes champs.

Il n'y a pas de contrainte d'unicité dans cette table.

Une colonne date permet de connaître la date d'archive.

reservation_hotel	hotels
id	: int(11) unsigned
libelle	: varchar(50)
etoile	: varchar(5)

reservation_hotel	histo_hotels
# id	: bigint(20) unsigned
libelle	: varchar(50)
etoile	: varchar(5)
date_archive	: timestamp

Le remplissage de la table historique peut se faire de manière automatique en utilisant les déclencheurs.

## Signal

Validation dans un trigger.

On peut utiliser les triggers pour vérifier la validité des données. On peut choisir de provoquer un signal sur un événement, si les nouvelles valeurs ne respectent pas certaines règles.

```
CREATE TRIGGER verif_etoiles
BEFORE INSERT ON hotels
FOR EACH ROW
BEGIN
    IF CHAR_LENGTH(NEW.nombre_etoiles) NOT BETWEEN 1 AND 5
    OR NEW.nombre_etoiles NOT REGEXP '^\\*{1,5}$'
    THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Le nombre d\'étoiles doit être entre 1 et 5, avec uniquement
des *';
    END IF;
END;
```

L'instruction **IF** en **SQL** doit se clôturer par **END IF;**

```
IF condition THEN
    Action;
END IF;
```

## CHECK


Une alternative au trigger signal, et d'utiliser le check. Valable depuis MySQL 8+

```
ALTER TABLE hotels
ADD CONSTRAINT chk_format_etoiles
CHECK (
    CHAR_LENGTH(nombre_etoiles) BETWEEN 1 AND 5
    AND nombre_etoiles REGEXP '^\\*{1,5}$'
);
```


## Test d'insertion

--  Valide

```
INSERT INTO hotels (nom, nombre_etoiles) VALUES ('Hôtel Paradis', '***');
```

--  Erreur (trop de \*)

```
INSERT INTO hotels (nom, nombre_etoiles) VALUES ('Hôtel Luxe', '*****');
```

--  Erreur (caractère interdit)

```
INSERT INTO hotels (nom, nombre_etoiles) VALUES ('Hôtel Problème', '*x*');
```

# Procédure stockée:

Une procédure stockée est un ensemble d'instructions SQL enregistré dans la base de données. Elle est appelée pour exécuter des actions répétitives.

## Syntaxe :

```
DELIMITER //
```

```
CREATE PROCEDURE nom_procedure(param1 TYPE, param2 TYPE)
```

```
BEGIN
```

```
    -- instructions SQL
```

```
END;
```

```
//
```

```
DELIMITER ;
```

Exemple :

```
DELIMITER //
```

```
CREATE PROCEDURE ajouter_utilisateur(nom VARCHAR(100), age INT)
```


```
BEGIN
```

```
    INSERT INTO utilisateurs(nom, age) VALUES(nom, age);
```

```
END;
```

```
//
```

```
DELIMITER ;
```

 Important : Très utile quand on a une insertion plus compliquée, par exemple qui utilise l'héritage ou qui doit vérifier une logique spécifique.

Utilisation

```
CALL ajouter_utilisateur('Alice', 30);
```

**CALL** permet d'exécuter une procédure stockée

# Fonction stockée

Une **fonction stockée** est comme une procédure, mais elle **retourne une valeur**. Elle peut être utilisée dans des requêtes SQL.

## Syntaxe

```
DELIMITER //  
CREATE FUNCTION nom_fonction(param TYPE)  
RETURNS TYPE  
DETERMINISTIC  
BEGIN  
    -- instructions SQL  
    RETURN valeur;  
END;  
//  
DELIMITER ;
```

## Exemple :

```
DELIMITER //  
CREATE FUNCTION calcul_tva(prix DECIMAL(10,2))  
RETURNS DECIMAL(10,2)  
BEGIN  
    RETURN prix * 0.20;  
END;  
//  
  
DELIMITER ;
```

## Utilisation :

```
SELECT calcul_tva(100); -- retourne 20
```

Une fonction est appelée dans un select.



# Les vues

Les vues sont des composants SQL qui encapsulent une requête SELECT dans des tables virtuelles nommées.

Une **vue** est une **représentation virtuelle** d'une requête. Elle facilite la réutilisation et sécurise l'accès aux données.

## Syntaxe

```
CREATE VIEW nom_vue AS  
SELECT ...  
FROM ...  
WHERE ...;
```

## Exemple

```
CREATE VIEW vue_utilisateurs_majeurs AS  
SELECT nom, age  
FROM utilisateurs  
WHERE age >= 18;
```

## Utilisation :

```
SELECT * FROM vue_utilisateurs_majeurs;
```

# Ressources

---

## 1. Sites de référence

- ❖ Documentation Officielle de MySQL : <https://dev.mysql.com/doc/refman/8.0/en/trigger-syntax.html>
- ❖ SQL.sh : <https://sql.sh/fonctions/agregation>
- ❖ W3 School : [https://www.w3schools.com/sql/sql\\_check.asp](https://www.w3schools.com/sql/sql_check.asp)

## 2. Tutoriels et Cours en Ligne

- ❖ TutorialsPoint : <https://www.tutorialspoint.com/sql/sql-transactions.htm>
- ❖ GeeksforGeeks : [https://www.geeksforgeeks.org/sql-injection/?ref=header\\_outind](https://www.geeksforgeeks.org/sql-injection/?ref=header_outind)

## 3. Exercices interactive :

- ❖ LeetCode : Problème à résoudre : <https://leetcode.com/problems/combine-two-tables/description/>
- ❖ SQLZoo : Exercice rapide : [https://www.sqlzoo.net/wiki/SQL\\_Tutorial](https://www.sqlzoo.net/wiki/SQL_Tutorial)
- ❖ LearnSQL - Contrainte CHECK : [LearnSQL CHECK Constraints](#)
- ❖ SQLBolt : [SQLBolt - Introduction to Views](#)