Les clés étrangères en pratique

Créer la table 'ticket' qui relie clients et événements via des clés étrangères. C'est le moment où le MCD prend vie dans la base : le SGBDR refusera désormais toute incohérence.

4TTR niveau

Au cours précédent, tu as créé `client` et `evenement`. Ce sont deux tables isolées : aucune ne sait qu'elle est liée à l'autre. C'est maintenant qu'on matérialise la relation `achète` / `concerne` du MCD avec une table `ticket` munie de **deux clés étrangères**.

🎯 Objectifs

À la fin de cette séquence, tu seras capable de :

1. Créer une table qui contient des clés étrangères
2. Déclarer une contrainte `FOREIGN KEY` dans HeidiSQL
3. Choisir un comportement `ON DELETE` / `ON UPDATE` adapté
4. Constater concrètement l'intégrité référentielle (le SGBDR refuse les incohérences)
5. Lire le `CREATE TABLE` complet généré avec ses contraintes

Partie 1 — Le rappel du MLD

```
client    (id, nom, prenom, email, telephone)
evenement (id, titre, date, prix_base)
ticket    (id, prix, statut, date_achat, #client_id, #evenement_id)
                                    ↳ client.id    ↳ evenement.id
```

La table `ticket` a deux clés étrangères, parce qu'elle participe à **deux relations** dans le MCD :

- `CLIENT 0,n — achète — 1,1 TICKET` → FK `client_id` côté ticket
- `TICKET 1,1 — concerne — 0,n ÉVÉNEMENT` → FK `evenement_id` côté ticket

C'est cohérent avec la règle du cours 07 : la FK va toujours côté `1,1`.

Partie 2 — Créer la table `ticket` (colonnes seulement)

Comme au cours précédent : clic droit sur la base `billetterie` → **Créer nouveau** → **Table**. Nom : `ticket`.

Les colonnes

Nom	Type	NULL	Note
`id`	`INT`	non	PK, AUTO_INCREMENT
`prix`	`DECIMAL(6,2)`	non	
`statut`	`VARCHAR(20)`	non	
`date_achat`	`DATETIME`	non	
`client_id`	`INT`	non	future FK
`evenement_id`	`INT`	non	future FK

Enregistre la table. Elle existe maintenant, mais `client_id` et `evenement_id` sont pour l'instant **de simples colonnes entières** : rien n'empêche d'y mettre `999` même si aucun client n°999 n'existe.

> 💡 Avant la prochaine étape, vérifie que les colonnes FK ont **EXACTEMENT LE MÊME TYPE** que la PK référencée. Toutes les deux doivent être `INT NOT NULL`. Si les types diffèrent, MySQL refusera de créer la contrainte.

Partie 3 — Ajouter les contraintes FOREIGN KEY

Ouvrir l'onglet des clés étrangères

Dans la fenêtre d'édition de la table `ticket`, va sur l'onglet **Clés étrangères** (ou **Foreign keys** selon ta langue). Tableau vide. Clique **Ajouter**.

Configurer la première FK : `client_id → client.id`

Champ	Valeur
Nom de la contrainte	`fk_ticket_client`
Colonnes	`client_id`
Table référencée	`billetterie.client`
Colonnes référencées	`id`
ON UPDATE	`CASCADE`
ON DELETE	`RESTRICT`

Configurer la seconde FK : `evenement_id → evenement.id`

Clique **Ajouter** à nouveau :

Champ	Valeur
Nom de la contrainte	`fk_ticket_evenement`
Colonnes	`evenement_id`
Table référencée	`billetterie.evenement`
Colonnes référencées	`id`
ON UPDATE	`CASCADE`
ON DELETE	`RESTRICT`

Enregistre. Si tout est bon, les deux contraintes apparaissent dans la liste.

Partie 4 — Que veulent dire `ON DELETE` et `ON UPDATE` ?

Ces options répondent à une question simple : **quand la ligne référencée bouge ou disparaît, qu'est-ce qui doit arriver aux lignes qui la référencent ?**

Les options

Option	Effet quand le client référencé est supprimé / modifié
`RESTRICT`	**Refuser** l'opération tant que des tickets référencent ce client
`CASCADE`	**Propager** : supprimer aussi les tickets / mettre à jour la valeur
`SET NULL`	Mettre `client_id` à `NULL` (seulement si la colonne autorise NULL)
`NO ACTION`	Équivalent à `RESTRICT` en MySQL

Le choix par défaut raisonnable

- `ON DELETE RESTRICT` — on **refuse** de supprimer un client qui a des tickets. Sinon, on perdrait silencieusement de l'historique de vente. Pour supprimer le client, il faudra d'abord traiter ses tickets explicitement.
- `ON UPDATE CASCADE` — si pour une raison étrange l'id d'un client changeait, on veut que les tickets suivent. En pratique, les `id AUTO_INCREMENT` ne changent jamais, mais cette option ne coûte rien.

> ⚠️ `CASCADE` sur `ON DELETE` peut sembler pratique ("je supprime le client, tout son historique part avec") mais c'est **dangereux** : une fausse manip efface des données qu'on ne voulait pas perdre. On l'utilise consciemment, jamais par défaut.

Partie 5 — Le SQL complet

Voici ce que HeidiSQL a envoyé à MySQL :

```sql
CREATE TABLE `ticket` (
    `id`           INT          NOT NULL AUTO_INCREMENT,
    `prix`         DECIMAL(6,2) NOT NULL,
    `statut`       VARCHAR(20)  NOT NULL,
    `date_achat`   DATETIME     NOT NULL,
    `client_id`    INT          NOT NULL,
    `evenement_id` INT          NOT NULL,
    PRIMARY KEY (`id`),
    CONSTRAINT `fk_ticket_client`
        FOREIGN KEY (`client_id`) REFERENCES `client` (`id`)
        ON UPDATE CASCADE ON DELETE RESTRICT,
    CONSTRAINT `fk_ticket_evenement`
        FOREIGN KEY (`evenement_id`) REFERENCES `evenement` (`id`)
        ON UPDATE CASCADE ON DELETE RESTRICT
);
```

Garde ce modèle sous les yeux : c'est exactement ce que tu écriras un jour à la main quand tu n'auras plus HeidiSQL sous la main.

Partie 6 — Tester l'intégrité référentielle

Le moment où le SGBDR montre **enfin** son utilité. Onglet **Requête**, exécute (F9) ces tests un par un.

Test 1 — Insertion valide

Vérifie d'abord les ids existants dans tes tables `client` et `evenement` (par exemple `SELECT id FROM client;`). Puis :

```sql
INSERT INTO ticket (prix, statut, date_achat, client_id, evenement_id)
VALUES (12.00, 'payé', '2026-10-01 14:30:00', 1, 1);
```

→ Succès. Le ticket est créé.

Test 2 — Client inexistant

```sql
INSERT INTO ticket (prix, statut, date_achat, client_id, evenement_id)
VALUES (12.00, 'payé', '2026-10-01 14:30:00', 999, 1);
```

→ **Erreur** :

```
Cannot add or update a child row: a foreign key constraint fails
```

Traduction : « Je refuse parce que ta clé étrangère pointe dans le vide. » Sans cette contrainte, tu te serais retrouvé avec un ticket orphelin — exactement le genre de bug silencieux qui pourrit une application en production.

Test 3 — Événement inexistant

```sql
INSERT INTO ticket (prix, statut, date_achat, client_id, evenement_id)
VALUES (15.00, 'réservé', '2026-10-01 15:00:00', 1, 999);
```

→ Même erreur, sur l'autre FK.

Test 4 — Supprimer un client référencé

```
DELETE FROM client WHERE id = 1;
```

→ **Erreur** :

```
Cannot delete or update a parent row: a foreign key constraint fails
```

C'est `ON DELETE RESTRICT` qui bloque : le client 1 a des tickets, on ne peut pas l'effacer sans gérer ses tickets d'abord. C'est exactement le comportement qu'on voulait.

> 💡 Pour vraiment supprimer le client, il faudrait d'abord supprimer ses tickets (`DELETE FROM ticket WHERE client_id = 1;`), puis le client. C'est plus de travail — c'est aussi le but : t'obliger à réfléchir avant de perdre des données.

Partie 7 — Visualiser les relations

HeidiSQL ne propose pas un vrai éditeur visuel de schéma, mais tu peux voir les relations en :

- Clic droit sur la table `ticket` → **Afficher CREATE TABLE** : tu vois toutes les contraintes en SQL
- Onglet **Clés étrangères** d'une table : la liste des FK sortantes

> 💡 Pour un schéma visuel propre, des outils comme
> MYSQL WORKBENCH
> ou **DBeaver** offrent une vue *Entity-Relationship*. Optionnel à ce stade — le SQL et le MLD papier suffisent.

🧪 Exercices

Exercice 1 — Insérer une vraie cohorte

Insère **5 tickets** valides, répartis entre tes clients et tes événements existants. Vérifie le résultat avec :

```
SELECT t.id, c.nom, e.titre, t.prix, t.statut
FROM ticket t
```

```
3   JOIN client c    ON c.id = t.client_id
4   JOIN evenement e ON e.id = t.evenement_id;
```

Tu n'as pas vu `JOIN` formellement — l'idée : la requête montre côte à côte les infos de trois tables grâce aux FK. C'est tout l'intérêt du modèle relationnel.

Exercice 2 — Casser intentionnellement

Tente d'insérer un ticket avec un `client_id` qui n'existe pas. Relève le message d'erreur exact. À quoi sert ce blocage dans une vraie application (billetterie en ligne par exemple) ?

Exercice 3 — Comprendre ON DELETE

Modifie la contrainte `fk_ticket_client` pour mettre `ON DELETE CASCADE`. Réessaie de supprimer le client n°1. Que se passe-t-il pour ses tickets ? Pourquoi est-ce dangereux dans la vraie vie ? Remets ensuite la valeur sur `RESTRICT`.

Exercice 4 — Ton projet

Ajoute toutes les **clés étrangères** manquantes de ton projet personnel. Pour chaque FK, justifie en une phrase ton choix de `ON DELETE` (`RESTRICT` ou `CASCADE`).

✍️ À retenir

- Une clé étrangère est une colonne déclarée comme `FOREIGN KEY (...) REFERENCES autre_table(id)`.
- La colonne FK doit avoir **exactement le même type** que la PK qu'elle référence.
- Le SGBDR **refuse** toute insertion ou mise à jour qui pointerait vers une ligne inexistante : c'est l'**intégrité référentielle**.
- `ON DELETE RESTRICT` est le défaut prudent ; `CASCADE` ne s'utilise que consciemment.
- Une table peut avoir **plusieurs** clés étrangères (autant que de relations auxquelles elle participe).
- HeidiSQL est un outil pratique ; le **SQL généré** est la source de vérité — apprends à le lire.

Suite

Tu as une base complète, propre, avec ses contraintes. Au prochain cours, on fait exactement le même travail dans **phpMyAdmin** — l'outil que tu utiliseras sur un hébergement web. Tu verras qu'au-delà de l'interface, ce sont les mêmes concepts et le même SQL en dessous.