

# INTRODUCTION AUX BASES DE DONNÉES AVEC SQL

**Enseignant :** Louis RAYNAL

**Contact :** [l-raynal@ices.fr](mailto:l-raynal@ices.fr)

**Public :** L3 Maths et M2 Bio-Santé

**Ressources :** <https://github.com/LouisRaynal/coursSQL>


**Établissement :** ICES

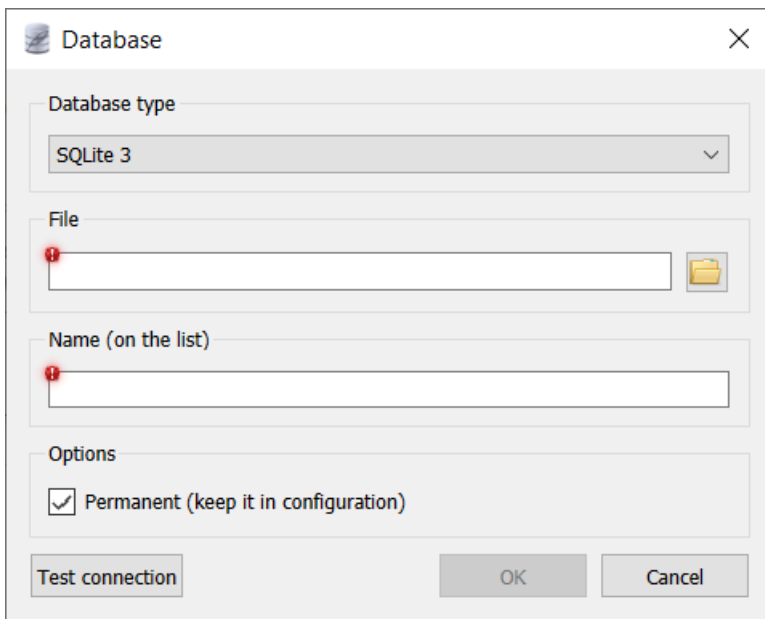
**Année :** 2025-2026

## Cours-TP 2 : Création d'une base de données

Nous allons reprendre l'exemple de la base de données bancaires utilisée dans le précédent cours-TP. L'objectif est de recréer cette base que l'on appellera **banque**, afin d'apprendre les différentes commandes de création et de modification d'un base.

### 1. Création d'une base (vide)

Dans SQLiteStudio, pour créer une nouvelle base de données SQLite vous n'avez qu'à cliquer sur  pour ouvrir l'outil de création de bases.



Il est possible avec  soit d'ouvrir une base existante, soit d'en créer une nouvelle.

#### Pratique

Créez une nouvelle base de données avec pour nom de fichier **banque.db** et avec pour nom de base **banque**. Stockez le fichier de base de données **sur le bureau** de votre ordinateur.

Sous Mac, vous devez d'abord sélectionner un répertoire de stockage, puis compléter le chemin d'accès (dans *File*) avec `/leNomDeVotreBase.db`

La base de données qui vient d'être créée est une base SQLite **vide**. Il faut ensuite créer les différentes tables constituant la base et les alimenter de données.

## 2. Création d'une table

Nous avons vu que lorsqu'une base de données suivait un modèle relationnel, l'élément principal de la base était la table.

En SQL, il est nécessaire de définir la **structure** d'une table au travers des **colonnes** la constituant. Chaque colonne va accueillir un certain **type de données** selon ce que l'on souhaite stocker dans la base.

Une table est aussi composée d'un **contenu**. Il s'agit de toutes les lignes de données qui vont venir remplir cette table. Chaque élément d'une ligne correspondra à une valeur de colonne (ou une valeur **NULL** si aucune n'est spécifiée).

Pour obtenir une table de données, on procède donc habituellement en deux temps :

1. **création de la structure** de la table, grâce à la commande **CREATE TABLE** ;
2. **insertion des données** dans la table créée grâce à la commande **INSERT INTO ... VALUES**.

### 2.1. Création de la structure d'une table

La syntaxe la plus basique pour la commande **CREATE TABLE** est la suivante :

#### Code 1 – Syntaxe basique de CREATE TABLE

```
1 CREATE TABLE nom_table
2 (
3     nom_col_1 type_col_1,
4     nom_col_2 type_col_2,
5     ...
6     nom_col_k type_col_k
7 )
8 ;
```

Le nom de la table est fourni après la commande **CREATE TABLE**. Il faut ensuite spécifier le nom de chaque colonne, ainsi que le type de données qu'elle accueillera.

Je vous recommande d'utiliser des noms concis mais clairs. Une première école consiste à utiliser uniquement des minuscules, avec chaque mot séparé par un `_`, par exemple : `nom_produit`. Une autre consiste à utiliser des noms de colonnes de la forme suivante : `nomProduit`. Il est aussi possible de spécifier des noms de colonnes en plusieurs mots en encadrant le nom par les symboles `" "`, `' '` ou `[ ]`.

#### Remarque

La casse dans les noms n'est en fait pas importante, mais il faut que vos colonnes aient des noms différents au sein d'une même table. Entre deux tables de la base, il est possible d'avoir plusieurs noms de colonnes identiques.

### Types de données

Spécifier un type de données permet de garantir une cohérence dans les informations stockées dans les tables d'une base de données. Parmi les types de données que vous rencontrerez le plus souvent il y a les **chaînes de caractères**, les **numériques** et les **données temporelles**. Dans chacune de ces trois catégories, vous trouverez différentes syntaxes SQL pour préciser ce que vous voulez que la colonne stocke. Nous ne présentons que les syntaxes les plus courantes, utilisables dans une base de données SQLite, et que nous serons amenés à rencontrer.

**Chaînes de caractères** - Dans la plupart des bases de données, vous trouverez majoritairement les syntaxes `VARCHAR(n)` et `CHAR(n)`. Une colonne avec ce type acceptera une chaîne de longueur maximale `n`, respectivement de longueur variable ou de longueur fixe. Une différence entre les deux est que `CHAR(n)` complétera la chaîne par des espaces à droite pour toujours atteindre la taille `n` définie. Pour des chaînes dont la longueur maximale est inconnue à l'avance nous pourrions utiliser la syntaxe `TEXT`.

**Numériques** - Pour définir une colonne qui contiendra des valeurs numériques, nous utiliserons la syntaxe :

- `INTEGER` lorsqu'il s'agit de valeurs entières ;
- `DECIMAL` lorsqu'il s'agit de valeurs décimales ;
- `BOOLEAN` lorsqu'il s'agit de valeurs 1 ou 0 (vrai/faux).

**Données temporelles** - Les dates et temps sont des données temporelles, habituellement saisies selon un format particulier.

- `'YYYY-MM-DD'` pour les dates, par exemple `'2022-05-15'` équivaut au 15/05/2022.  
On utilisera alors la syntaxe `DATE`.
- `'HH:MM:SS'` pour les temps, par exemple `'16:41:47'` équivaut à 16 heures 41 minutes et 47 secondes.  
On utilisera alors la syntaxe `TIME`.
- `'YYYY-MM-DD HH:MM:SS'` pour les dates/temps, par exemple `'2022-05-15 16:41:47'`.  
On utilisera alors la syntaxe `DATETIME`.

### Attention

Pour la saisie de chaînes de caractères, il faut écrire une chaîne entre `' '`. Par exemple, `'Blanc'`.

Pour la saisie de données numériques, **il ne faut pas utiliser de `' '`**. Les nombres décimaux s'écrivent avec un `.` par exemple `3.14`.

Pour la saisie de données temporelles, il faut les écrire entre `' '`. Par exemple, `'2022-05-15'`.

## Contraintes de colonne et de table

Lors de la création d'une table, nous pouvons définir des **contraintes** sur une colonne ou un ensemble de colonnes. Une contrainte est une règle visant à limiter les données pouvant être insérées dans une table, afin de garantir l'exactitude et la fiabilité des données de la base de données. Si une contrainte n'est pas vérifiée lorsque vous essayez d'ajouter de nouvelles données, un message d'erreur sera retourné.

La syntaxe précédente se complexifie de la manière suivante :

### Code 2 – Syntaxe de CREATE TABLE avec contraintes

```
1 CREATE TABLE nom_table
2 (
3     nom_col_1 type_col_1 contrainte_col_1,
4     nom_col_2 type_col_2 contrainte_1_col_2 contrainte_2_col_2,
5     nom_col_3 type_col_3,
6     contrainte_table_1,
7     contrainte_table_2
8 )
9 ;
```

Les **contraintes de colonne** sont renseignées après le type de la colonne, grâce à des mots-clés, et ne concernent que la colonne après laquelle elles sont listées. Les contraintes les plus courantes sont :

- **NOT NULL** : pour garantir que la colonne ne contienne pas de valeurs NULL. Sans cette contrainte, si une ligne de données est insérée dans la table, et qu'aucune valeur n'est renseignée pour une certaine colonne, alors une valeur vide (NULL) sera utilisée ;
- **DEFAULT** : pour définir une valeur par défaut à utiliser lorsqu'aucune n'est spécifiée lors de l'ajout d'une ligne de données. Il faut faire suivre le mot clé **DEFAULT** de la valeur à utiliser par défaut ;
- **UNIQUE** : pour garantir que toutes les valeurs dans la colonne sont différentes ;
- **PRIMARY KEY** : pour définir la colonne comme une clé primaire et ainsi identifier de manière unique chaque ligne de la table ;
- **CHECK** : pour garantir que toutes les valeurs d'une colonne satisfont certaines conditions listées après ce mot clé.

Les **contraintes de table** sont renseignées à la fin de la requête. Elles fonctionnent comme les contraintes de colonne, sauf qu'elles peuvent concerner plusieurs colonnes qu'il est nécessaire de nommer.

Par exemple

```
PRIMARY KEY (nom_col_1, nom_col_2, nom_col_3)
```

définira une clé primaire (composite) basée sur les valeurs jointes de ces trois colonnes.

D'une manière similaire vous pouvez écrire

```
UNIQUE (nom_col_1, nom_col_3)
```

pour garantir l'unicité jointe des valeurs de ces deux colonnes.

Vous pouvez utiliser **CHECK** sur plusieurs colonnes à la fois.

```
CHECK (nom_col_1 > nom_col_3)
```

permettrait de garantir que les valeurs de **nom\_col\_1** sont strictement supérieures à celles de **nom\_col\_3**.

Une dernière contrainte de table importante est la **définition d'une clé étrangère**. Cette contrainte s'écrit de la manière suivante :

```
FOREIGN KEY (fk_nom_col) REFERENCES table_mère (nom_col)
```

**FOREIGN KEY** est suivi de la (ou les) colonne(s) formant la clé étrangère. Il faut ensuite indiquer où se trouve la clé primaire correspondante, via la commande **REFERENCES** qui est suivi du nom de la table hébergeant la clé primaire (**table\_mère**), et le nom de la clé primaire dans cette même table.

### Pratique




Revenons maintenant à notre exemple de données bancaires afin d'appliquer ce que nous venons de voir (reprenez votre TP1 si besoin).

La table **clients** est composée de trois colonnes avec les caractéristiques suivantes :

- **id\_client**, qui accueillera des valeurs entières (type **INTEGER**) et sera une clé primaire (**PRIMARY KEY**) pour cette table ;
- **nom**, qui accueillera des chaînes de caractères (prenons par exemple le type **VARCHAR** avec 100 caractères maximum), et sera toujours renseignée (**NOT NULL**) ;

- **prénom**, qui accueillera des chaînes de caractères (**VARCHAR(100)**), et sera toujours renseignée (**NOT NULL**).

Étant donné ces informations, créons la table **clients** en utilisant la requête ci-dessous.

Si ce n'est pas déjà fait, commencez par ouvrir un éditeur SQL avec le bouton  et sauvegardez votre script avec  (en faisant attention à avoir une extension **.sql**). Réécrivez ensuite là requête ci-dessous dans la partie "Requête", puis exécutez là avec le bouton .

#### Code 3 – Création de la table **clients**

```

1 CREATE TABLE clients
2 (
3     id_client INTEGER PRIMARY KEY,
4     nom       VARCHAR(100) NOT NULL,
5     prénom    VARCHAR(100) NOT NULL
6 )
7 ;

```

La table **clients** devrait apparaître sur la gauche, comme table de la base de données **banque**.

#### Attention

Si vous remarquez qu'une table que vous avez créée contient une erreur (par exemple dans l'orthographe du type d'une colonne), il faudra dans un premier temps la supprimer, puis dans un second temps la recréer en exécutant votre requête (corrigée) de création de la table.

Pour supprimer une table, utilisez la commande

```
DROP TABLE nom_table ;
```

Il n'y a pas de retour en arrière possible. Cela supprimera la structure de la table et tout son contenu.

#### Remarque

Lorsque vous écrivez une requête SQL voici quelques conseils pour faciliter sa (re)lecture et éviter des erreurs.

1. **Utiliser des indentations.** Il est possible d'écrire une requête complète sur une seule ligne, mais ce serait difficilement compréhensible. Je vous recommande plutôt d'indenter (via des retours à la ligne et la touche Tabulation) certaines parties de vos codes pour les rendre plus lisibles. Vous pouvez aussi utiliser le formateur de code SQL avec l'icône **T**.
2. **Commenter vos requêtes.** Vous pouvez ajouter des lignes de commentaires, qui ne seront pas exécutées mais qui vous permettront de documenter vos requêtes. Vous pouvez soit faire un commentaire d'une ligne avec deux symboles - successifs, soit un commentaire multi-lignes avec **/\* \*/**.

```

/* Ceci est un commentaire
multi-lignes */

-- Ceci est un commentaire d'une ligne

```

3. **Terminer vos requêtes par un ;.** Cela n'est pas obligatoire, mais cela vous permettra de pouvoir sélectionner et exécuter plusieurs requêtes à la fois.

## Pratique

Générons maintenant la structure de la table `comptes` dont les colonnes sont :

- `id_cpt`, qui est de type `INTEGER` ainsi qu'une clé primaire.
- `type_cpt`, qui est de type `VARCHAR(100)` à valeurs non vides. Pour cette colonne nous pouvons ajouter une contrainte pour vérifier que le type de compte est soit égal à `'épargne'`, soit à `'courant'`.
- `fk_id_client`, qui est de type `INTEGER` et une clé étrangère (la clé primaire correspondante se trouve dans la table `clients`, colonne `id_client`).
- `solde`, qui est de type `DECIMAL`, à valeurs non vides, avec comme valeur par défaut 0.

### Code 4 – Création de la table `comptes`

```

1 CREATE TABLE comptes
2 (
3     id_cpt          INTEGER          PRIMARY KEY,
4     type_cpt        VARCHAR(100)    NOT NULL
5                                     CHECK(type_cpt IN ('épargne', 'courant')),
6     fk_id_client    INTEGER          NOT NULL,
7     solde           DECIMAL          DEFAULT(0)    NOT NULL,
8     FOREIGN KEY (fk_id_client)      -- identité de la clé étrangère
9     REFERENCES clients (id_client)  -- liée à la col. id_client de la table clients
10    -- Notez que ces deux dernières lignes forment une seule contrainte de table
11    -- sinon elles auraient été séparées par une virgule.
12    -- Vous pouvez les écrire sur une même ligne si vous le souhaitez.
13 )
14 ;

```

## Pratique

Écrivez une requête pour créer la structure de la table `transactions`.

## 3. Insertion de données dans une table

Une table nouvellement créée ne contient aucune information. Il faut la remplir en indiquant les valeurs de lignes à insérer dans la table. Pour ce faire on utilise la commande `INSERT INTO ... VALUES` qui a pour syntaxe :

### Code 5 – Syntaxe de `INSERT INTO ... VALUES` pour insérer une ligne de données

```

1 INSERT INTO nom_table (nom_col_1, nom_col_2)
2 VALUES (val_1_col_1, val_1_col_2)
3 ;

```

Une seule ligne de données sera ainsi insérée grâce à cette syntaxe. Les valeurs spécifiées (`val_1_col_1`, `val_1_col_2`) viendront remplir les colonnes indiquées (`nom_col_1`, `nom_col_2`) de la table `nom_table`.

## Remarque

Vous pouvez insérer plusieurs lignes de données à la fois en listant les valeurs les unes à la suite des autres.

### Code 6 – Syntaxe de INSERT INTO ... VALUES pour insérer plusieurs lignes de données

```
1 INSERT INTO nom_table (nom_col_1, nom_col_2)
2 VALUES
3     (val_1_col_1, val_1_col_2),
4     (val_2_col_1, val_2_col_2),
5     (val_3_col_1, val_3_col_2)
6 ;
```

## Pratique

Alimentons maintenant les tables `clients`, `comptes` et `transactions` de lignes de données.

Pour la table `clients`, voici le code à utiliser, qui contient trois requêtes car nous voulons insérer trois lignes :

### Code 7 – Insertion des données clients

```
1 INSERT INTO clients (id_client, nom, prénom) VALUES (1, 'Blanc', 'Clémence');
2 INSERT INTO clients (id_client, nom, prénom) VALUES (2, 'Dumont', 'Charles');
3 INSERT INTO clients (nom, prénom) VALUES ('Blake', 'George');
```

Notez qu'il n'est en fait pas nécessaire de remplir les valeurs des clés primaires, elles s'incrémentent automatiquement en se basant sur la plus grande valeur déjà présente.

Pour les tables `comptes` et `transactions`, à vous de rédiger les requêtes d'insertion de données en utilisant les données suivantes :

### Code 8 – Données à insérer dans les tables comptes et transactions

```
1 -- Pour comptes
2 (101, 'épargne', 1, 500.00)
3 (102, 'courant', 1, 250.00)
4 (103, 'courant', 2, 300.00)
5 (104, 'épargne', 3, 0.00) -- Note : DEFAULT(0) nous autorise à ne pas insérer ce 0
6 (105, 'courant', 3, 1000.00)
7
8 -- Pour transactions
9 (501, 'débit', 102, 55.00, '2022-01-02')
10 (502, 'crédit', 105, 10000.00, '2022-02-05')
11 (503, 'débit', 105, 9000.00, '2022-02-06')
12 (504, 'débit', 103, 200.00, '2022-02-08')
13 (505, 'crédit', 101, 500.00, '2023-03-09')
```

## 4. Mise à jour de données dans une table

Il est parfois nécessaire d'actualiser les données, par exemple en cas d'erreurs de saisie avec la commande `INSERT INTO`. Pour cela il existe la commande `UPDATE` pour remplacer des valeurs existantes par d'autres valeurs. En voici la syntaxe

### Code 9 – Syntaxe de UPDATE

```
1 UPDATE nom_table
2 SET nom_col_1 = new_val_col_1,
3   nom_col_2 = new_val_col_2
4 WHERE condition
5 ;
```

Toutes les lignes vérifiant une `condition`, à définir, auront leurs valeurs actuelles des colonnes `nom_col_1` et `nom_col_2` remplacées par `new_val_col_1` et `new_val_col_2` respectivement.

`condition` effectue une **comparaison logique** sur **chaque ligne de la table**, afin d'identifier quelles sont les lignes qui seront mises à jour.

### Remarque

Une **condition** est une combinaison de divers expressions SQL qui s'évalue à une valeur booléenne (1 sur les lignes pour lesquelles cette condition est vérifiée, 0 sinon).

Nous en verrons plus sur les **conditions** dans le TP 3.

Dans la requête ci-dessus, l'usage le plus commun de la `condition` consiste à identifier les lignes à changer en se basant sur une valeur unique d'une colonne, par exemple une clé primaire !

### Pratique

Dans la table `transactions`, remarquez que l'une des dates est fausse. La date de la transaction `id_tran=505` devrait être non pas '2023-03-09' mais '2022-03-09'. Nous pouvons effectuer la mise à jour de la manière suivante :

### Code 10 – Exemple d'UPDATE

```
1 UPDATE transactions
2 SET date = '2022-03-09'
3 WHERE id_tran = 505
4 ;
```

Lorsque vous mettez à jour une table, vérifiez bien qu'elle a été modifiée comme attendu. Par exemple avec le code suivant qui permet d'afficher tout le contenu de la table `transactions`.

```
SELECT *
FROM transactions
;
```

Pour vous exercer, effectuez les mises à jour listées ci-dessous.

1. Changer le nom de Clémence 'Blanc' en 'Blake'.
2. Mettre le solde du compte courant de George Blake à 0.
3. Réduire le montant de toutes les transactions de type 'débit' de 10 euros.



## 5. Suppression de lignes dans une table

Dans une table, il est possible de supprimer des lignes vérifiant une certaine **condition** à définir. Pour cela nous utiliserons la commande **DELETE**, dont la syntaxe est

### Code 11 – Syntaxe de DELETE

```
1 DELETE FROM nom_table
2 WHERE condition
3 ;
```

### Attention

Si vous oubliez la partie **WHERE condition**, ou indiquez une condition qui est toujours vraie (comme **WHERE 1**) alors vous supprimerez toutes les lignes de votre table !

Cela est aussi vrai pour les requêtes de mise à jour.

### Pratique

Si l'on veut par exemple supprimer toutes les lignes de la table **clients** avec comme nom 'Blake', nous pouvons utiliser la commande :

### Code 12 – Exemple de DELETE

```
1 DELETE FROM clients
2 WHERE nom = 'Blake'
3 ;
```

La syntaxe est correcte, mais **cela ne fonctionnera pas sur notre exemple** ! Vous aurez le message *“Erreur pendant l'exécution de la requête sur la base de données « banque » : FOREIGN KEY constraint failed”*.

Le problème est que l'on souhaite supprimer des lignes de la table **clients**, donc aussi des valeurs de la clé primaire **id\_client**. Or cette clé primaire est liée à la clé étrangère **fk\_id\_client** de la table **comptes**. (Ce lien a été défini par la contrainte de table **FOREIGN KEY** lors de la création de **comptes**).

Le comportement par défaut de SQLite est d'empêcher la suppression de valeurs d'une clé primaire qui pourrait générer des incohérences entre les tables. Si vous supprimez les clients avec pour nom 'Blake', vous perdrez aussi leurs **id\_client**. Il y aurait alors des lignes dans la table **comptes** avec des identifiants de clients qui n'existent plus !

Lors de la création de la structure d'une table (commande **CREATE TABLE**), il est possible de définir des comportements à adopter lors de la suppression de valeurs d'une clé primaire. Rajouter **ON DELETE CASCADE** lors de la définition de la clé étrangère, permet d'autoriser la suppression en cascade des lignes utilisant les valeurs de clé supprimées.

### Code 13 – Exemple d'une contrainte de table FOREIGN KEY avec ON DELETE CASCADE

```
1 ...
2 FOREIGN KEY (fk_id_client)
3 REFERENCES clients (id_client)
4 ON DELETE CASCADE
5 ...
```

D'une manière similaire, nous pouvons mettre à jour en cascade une clé étrangère en ajoutant

```
ON UPDATE CASCADE
```

De cette manière la modification d'une valeur de clé primaire via la commande `UPDATE`, se répercutera aussi sur les clés étrangères associées.

## 6. Exercices

### Pratique

Votre objectif est de créer une base de données pour une bibliothèque souhaitant stocker des informations sur ses *abonnés*, sur ses *livres* et sur les *emprunts* qui sont faits.

Il y a trois types d'entités qui vous intéressent et sur lesquels vous voudrez récupérer des informations : *abonné*, *livre* et *emprunt*. Vous devrez donc créer trois tables, que l'on appellera respectivement **abonnés**, **livres** et **emprunts**. On nommera la base de données **bibliothèque**.

Sur un *abonné*, on souhaite récupérer les informations suivantes :

- numéro d'abonné
- nom
- prénom
- numéro de téléphone
- code postal

Sur un *livre*, les informations suivantes :

- titre
- auteur
- année d'édition
- prix

Sur un *emprunt*, les informations suivantes :

- identité de l'abonné emprunteur
- identité du livre emprunté
- date d'emprunt
- date de retour (lorsque connue)

1. Complétez la liste des informations à recueillir afin que chaque table possède une clé primaire.
2. Pour chaque information, décidez d'un nom de colonne simple qui sera utilisé dans les tables de la base de données.
3. Décidez aussi du type de chaque colonne, selon les données à stocker.
4. Réfléchissez aux potentielles contraintes de colonne à poser.
5. Identifier les informations qui seront des clés étrangères dans les tables de votre base.
6. Dessinez le schéma relationnel de votre base **bibliothèque**.

7. Créez une base de données vide nommée **bibliothèque**, qui sera stockée dans un fichier **bibliothèque.db**.
8. Rédigez des requêtes SQL afin de créer la structure des tables **abonnés**, **livres** et **emprunts** en utilisant vos réponses aux questions précédentes.
9. Rédigez des requêtes d'insertion de données afin d'inclure trois livres de votre choix dans la table **livres**.
10. Rédigez des requêtes d'insertion de données afin de vous ajouter comme abonné de la bibliothèque, ainsi que votre voisin de gauche et/ou de droite.
11. Vos voisins souhaitent chacun emprunter un livre dans la bibliothèque, rédigez des requêtes d'insertion de données afin de stocker les informations sur ces emprunts. Laissez vide la date de retour pour chaque emprunt.
12. Rédigez une requête de mise à jour de données afin de préciser la date de retour des livres.
13. Rédigez une requête de suppression de lignes afin de supprimer un livre non emprunté de la bibliothèque, et une autre pour supprimer un emprunt.
14. Rédigez une requête de suppression de la table **emprunts**, puis **livres** et enfin **abonnés**.