

Utiliser une base données locale : SQLite

Table des matières

I. Contexte	3
II. Qu'est-ce que SQLite ?	3
III. Exercice : Appliquez la notion	6
IV. Utiliser SQLite dans son projet	6
V. Exercice : Appliquer la notion	15
VI. Essentiel	17
VII. Auto-évaluation	18
A. Exercice final	18
B. Exercice : Défi.....	19
Solutions des exercices	19

I. Contexte

Durée : 1 h

Environnement de travail : Une application React Native initialisée ou utiliser <https://snack.expo.io/>

Pré-requis : Bases de React Native et SQL

Contexte

Lorsque l'on utilise une application mobile, il peut être pertinent de stocker certaines données qui doivent persister entre les différents cycles de vie de l'application.

En effet, tant que l'application vit en mémoire, elle oscille entre un cycle *background* (arrière-plan), *foreground* (premier-plan) et *killed* (tué : le processus a été arrêté et la mémoire purgée).

Il est parfois dommage de perdre le contenu d'un état lorsque l'application est arrêtée, par exemple une liste de tâches à faire. Pour remédier à cela, on peut utiliser une base de données locale telle que SQLite, afin de conserver les données dans le téléphone et les réutiliser plus tard.

On profite également d'un système de requêtes performant pour interagir avec ces données grâce à la syntaxe SQL.

II. Qu'est-ce que SQLite ?

Objectif

- Comprendre ce qu'est une base de données relationnelle

Mise en situation

SQLite est une version embarquée d'un système de gestion de bases de données relationnelles, qui permet à une application d'interagir avec une faible latence avec un moteur de stockage.

Le principe est que les données sont stockées dans un fichier écrit directement sur le terminal où l'application est installée, et ce fichier pourra ensuite être interrogé avec un langage de requête : SQL.

Les bases de la structure de données en SQL

Dans le monde des bases de données, les plus répandues sont les bases de données relationnelles. Le principe étant de définir une structure de données fortement typée, qui permet de ranger les valeurs à stocker.

Ces valeurs peuvent posséder des relations entre elles, et c'est le système de gestion de base de données qui permet de garantir une intégrité entre ces enregistrements. Ainsi, dès lors qu'une relation existe, si, par exemple, on supprime un de ses maillons, cela sera répercuté sur les éléments qui lui étaient liés.

Par exemple, si on définit qu'un utilisateur possède une liste de tâches à traiter, il est possible de déclarer une règle indiquant qu'un utilisateur ne peut être supprimé de la base de données que si l'on supprime les tâches qui lui sont liées auparavant. Cette sécurité permet de garantir l'intégrité de la base et la non-existence d'enregistrements orphelins.

Design de tables en SQL

Pour déclarer le schéma de données vu plus haut, nous utilisons une entité nommée `Tables`, qui permet de ranger les données selon ce qu'elles sont. Les deux tables ci-dessous sont étroitement liées.

```

1 // Table Utilisateur
2 | id | prenom | nom |
3 |----|-----|-----|
4 | 1 | Andréas | HANSS |
5 | 2 | Claude | Delatour |
6
7 // Table Tâches
8 | id | owner | nom | done |
9 |----|-----|-----|-----|
10 | 1 | 1 | Faire un cours React Native | false |
11 | 2 | 1 | Faire de la veille | true |
12 | 3 | 2 | Passer l'aspirateur | true |

```

Chaque ligne (enregistrement) est identifiée par une valeur unique dans la table, ici `id` : on parle alors de **clé primaire**.

Cette clé permet de facilement identifier et récupérer un enregistrement, elle garantit également que la valeur est unique dans la table. Ici, la clé primaire est simple, mais on pourrait utiliser une clé composée de plusieurs champs si on le souhaite. Par exemple, pour un système de notation de recettes de cuisine, on aurait une clé primaire composée de l'`id` de l'utilisateur et de l'`id` de la recette, ce qui garantit que le couple utilisateur/recette est unique dans la base de données.

Dans cet exemple, la table `Utilisateur` possède deux enregistrements (à savoir que les champs dans une table sont représentés sous forme de colonnes).

Les champs peuvent être optionnels ou obligatoires, mais chaque enregistrement dans une table a le même nombre de colonnes.

Dans la deuxième table, on définit la liste des tâches à effectuer, qui sont également identifiées par une clé primaire nommée `id`, leur nom et leur statut de complétion. Mais, surtout, par une relation vers la table `Utilisateur` grâce au champ `owner` qui contient l'`id` de l'utilisateur à qui appartient la tâche.

Ce type de relation est nommée une **clé étrangère** et permet de garantir que l'identifiant passé dans cette colonne de la table soit obligatoirement un identifiant qui existe dans la table ciblée. Le cas échéant, une erreur est déclenchée.

Ces notions sont fondamentales au langage SQL. Nous les utiliserons souvent.

Définir la structure de nos tables

Pour créer une table, on utilise le langage SQL. Voici un exemple pour les tables définies précédemment :

```

1 CREATE TABLE Users (
2   id INTEGER PRIMARY KEY,
3   prenom TEXT NOT NULL,
4   nom TEXT NOT NULL
5 );
6 CREATE TABLE Tasks (
7   id integer PRIMARY KEY,
8   owner_id integer,
9   nom TEXT NOT NULL,
10  done BOOLEAN,
11  FOREIGN KEY (owner_id) REFERENCES Users (id)
12 );

```

On utilise différents mots-clés ici : `CREATE TABLE` sert à définir la structure des tables nommées `Users` et `Tasks`. On spécifie ensuite chaque champ nécessaire à la structure des données représentées, en indiquant tout d'abord son nom, puis son type, et enfin ses contraintes si nécessaire. De cette manière, ici, on définit des clés primaires, `id`, avec `PRIMARY KEY` et une clé étrangère avec `FOREIGN KEY (owner_id) REFERENCES Users (id)`. Cette contrainte indique que le champ `owner_id` **réfère** le champ `id` de la table `Users`. La contrainte `NOT NULL` spécifie que le champ est obligatoire lors de l'ajout d'un nouvel enregistrement. Par défaut, lorsqu'un champ n'est pas renseigné, sa valeur est mise à `NULL`.

Dans SQLite, les types de données utilisables sont les suivants :

Classe de stockage	Signification
NULL	Valeur manquante ou inconnue
INTEGER	Nombre entier positif ou négatif de 1, 2, 3, 4 ou 8 octets
REAL	Nombre décimal à virgule de 8 octets
TEXT	Un texte avec encodage au choix et longueur illimitée
BLOB	BLOB signifie <i>binary large object</i> et ils sont utilisés pour stocker n'importe quel type de données Leur taille théorique est illimitée

Grâce à cela, on peut définir et composer comme on le souhaite le format des données de nos différentes entités dans des tables.

Remarque Le principe des transactions

Avant de commencer l'installation et les exemples, définissons le principe de transactions en SQL.

Étant donné que les données ont une forte relation entre elles et qu'on veut garantir leur intégrité, SQL met à disposition un système de transaction, qui permet de figer l'état de la base de données le temps qu'on exécute un certain nombre d'opérations.

Dans l'exemple ci-dessous, très minimaliste, imaginons que des utilisateurs possèdent chacun un compte en banque avec de l'argent, et qu'ils souhaitent pouvoir s'échanger des fonds.

```

1 // Table Utilisateur
2 | id | username | money |
3 |----|-----|-----|
4 | 1 | andreas | 100   |
5 | 2 | hugo    | 0     |

```

Pour réaliser un échange monétaire de cinquante euros, par exemple, il faut enlever cinquante euros à un des utilisateurs, puis en rajouter cinquante à l'autre.

La première chose à faire est de récupérer l'état actuel des comptes, puis de procéder à la modification.

Imaginons maintenant qu'après avoir enlevé cinquante euros au premier, l'application plante avant que l'on ait pu ajouter cinquante euros au second. Le premier n'aurait donc plus que cinquante euros et le deuxième serait toujours à zéro : il y a donc un risque de perte de données.

Sans transaction, il n'est pas possible de figer l'état de la base de données afin de garantir l'exécution de l'ensemble des instructions. On peut donc se retrouver dans certains cas avec des incohérences dans les données.

En utilisant les transactions, on sécurise les opérations en garantissant que toutes les instructions s'exécutent dans un contexte défini, au sein duquel l'intégrité des valeurs est garantie. Si une erreur devait intervenir, la transaction serait automatiquement relancée ou abandonnée.

Utilisation avec React Native

SQLite est disponible sous forme de librairie pour Android et iOS et donc, par extension, pour Expo avec React Native. Pour cela, un module « pont natif » spécifique pour l'utilisation de SQLite est disponible.

Si on se trouve sur la version installée d'Expo sur son ordinateur, il faut lancer la commande `expo install expo-sqlite`, et faire l'import ci-dessous dans un fichier.

Pour la version web, il n'est pas nécessaire de taper la commande.

```
import * as SQLite from 'expo-sqlite';
```

Syntaxe À retenir

- SQLite est une base de données relationnelle embarquée utilisable dans React Native. L'utilité est de pouvoir persister des données entre les cycles de vie de l'application, par exemple, lorsque l'on redémarre son téléphone portable.
- C'est très pratique, notamment pour les applications hors-ligne ou la mise en cache de données pour éviter de requêter le réseau et donc économiser de la bande passante.
- Les données sont définies dans des tables et s'identifient entre elles par le biais de clés primaires et clés étrangères.
- Pour modifier les données, on utilise des transactions afin de garantir l'intégrité de la base.

Complément

- <https://sql.sh/>

III. Exercice : Appliquez la notion

Question

[solution n°1 p.21]

En vous basant sur ce que nous avons vu précédemment, définissez un modèle de données pour gérer une petite application de liste de cuisine.

Les utilisateurs possèdent un nom, un prénom et un mail, et peuvent partager leurs recettes de cuisine.

Les recettes de cuisine ont un titre, une description et chaque utilisateur peut noter la recette de cuisine.

Une fois le modèle défini, essayez de générer les instructions qui permettent de créer les tables.

Pour marquer une colonne comme ayant au maximum un enregistrement avec cette valeur, on peut utiliser le mot-clé `UNIQUE`.

IV. Utiliser SQLite dans son projet

Objectif

- Réaliser une mini application avec SQLite

Mise en situation

Dans ce chapitre, nous allons prendre comme exemple une petite boutique avec des produits à la vente et un stock, l'utilisateur pourra également alimenter un panier d'achat.

Nous allons voir comment interagir avec la liste des produits dans un premier temps, puis, plus tard, nous câblerons le panier d'achat à cette liste de produits dans un exercice.

Pour commencer, il faut d'abord définir le modèle de données, insérer des éléments dans la base de données, les récupérer et gérer l'état du panier.

Notre petite boutique

Commençons par exprimer le besoin.

Nous voulons une boutique qui gère un ensemble de produits, chaque produit possède :

- un nom
- une description
- une quantité de stock

Ces produits peuvent être achetés par des utilisateurs qui sont définis par :

- nom
- prénom
- e-mail

Pour acheter, l'utilisateur possède une ou plusieurs listes de souhaits, qui sont en fait des paniers où il va ranger des produits pour pouvoir les commander :

- un nom de panier (ou liste de souhaits)
- et une liste d'identifiants de produits triés par ordre

Modélisons tout cela :

```
1 CREATE TABLE IF NOT EXISTS Users (  
2   id INTEGER PRIMARY KEY,  
3   first_name TEXT NOT NULL,  
4   last_name TEXT NOT NULL,  
5   email TEXT NOT NULL UNIQUE  
6 );  
7 CREATE TABLE IF NOT EXISTS Products (  
8   id INTEGER PRIMARY KEY,  
9   title TEXT NOT NULL,  
10  description TEXT NOT NULL,  
11  quantity INTEGER  
12 );  
13 CREATE TABLE IF NOT EXISTS WishLists (  
14   id INTEGER PRIMARY KEY,  
15   owner_id INTEGER,  
16   title TEXT NOT NULL,  
17   creation_date INTEGER NOT NULL,  
18   FOREIGN KEY (owner_id) REFERENCES Users (id) ON DELETE CASCADE  
19 );  
20 CREATE TABLE IF NOT EXISTS WhishList_Products (  
21   whishlist_id INTEGER,  
22   product_id INTEGER,  
23   list_order INTEGER,  
24   PRIMARY KEY (wishlist_id, product_id, list_order),  
25   FOREIGN KEY (wishlist_id) REFERENCES WishLists (id) ON DELETE CASCADE,
```

```
26 FOREIGN KEY (product_id) REFERENCES Products (id) ON DELETE CASCADE
27 );
```

Notons qu'il n'existe pas de type `Date` en SQLite : on utilise donc un entier où l'on va stocker un timestamp UNIX. Pour plus d'informations, rendez-vous ici : <https://www.sqlitetutorial.net/sqlite-date/>.

Avant de passer à la suite, sortons un peu la tête du cours et voyons pourquoi et comment créer un Hook Personnalisé.

Le Hook personnalisé

Construire notre propre Hook va nous permettre d'extraire la logique d'un composant sous forme de fonctions réutilisables. Imaginons vouloir monter une application faisant appel à une API. Cette API contient différents types de données.

Appeler l'API entière demanderait trop de ressources à notre application et la ralentirait de manière significative. Pour des soucis de maintenabilité dans le temps et d'optimisation, nous allons alors créer notre propre Hook grâce à la même logique que nous allons intégrer à une fonction.

Est-ce que vous vous imaginez devoir reproduire le même code, plusieurs fois, pour exactement la même fonctionnalité ? (Ici faire un axios ou un fetch afin d'avoir des données à notre disposition...)

Dans notre cas, nous allons vouloir effectuer plusieurs fois une requête en utilisant fetch. (Notez toutefois que nous garderons un exemple volontairement "simple" avec le fetch pour que ce soit plus facilement compréhensible. **Si Fetch vous semble flou ou encore énigmatique, référez-vous au cours Les promesses et l'API Fetch dans le thème Dynamisez vos sites web avec Javascript !**)

Nous appellerons notre Hook `useFetch`, tout simplement : Nous utiliserons l'API JSON Placeholder disponible gratuitement [ici](https://jsonplaceholder.typicode.com/)¹. Notez toutefois que, dans mon exemple, nous allons utiliser une seule et même API, mais il n'est pas exclu que vous ayez, par exemple, à en utiliser plusieurs. Lors d'un fetch, il y a toute une structure à respecter pour pouvoir appeler l'API. Vous en trouverez la structure ci-dessous :

```
1 import React, { useEffect, useState } from 'react';
2 import { FlatList, StyleSheet, Text, View } from 'react-native';
3
4 function Posts() {
5   const [posts, setPosts] = useState([])
6   useEffect(function () {
7     (async function () {
8       const response = await fetch('https://jsonplaceholder.typicode.com/posts?_limit=10')
9       const responseData = await response.json()
10      if(response.ok) {
11        setPosts(responseData)
12      } else {
13        alert(JSON.stringify(responseData))
14      }
15    }) ()
16  }, [])
17
18  return <FlatList>
19    {posts.map(post => <Text>{post.title}</Text>)}
20  </FlatList>
21 }
22
23 function App() {
24
25   return (
26     <View style={styles.app}>
27       <Posts />
```

¹ <https://jsonplaceholder.typicode.com/>


```

28     </View>
29   );
30 }
31
32 export default App;

```

Comme vous pouvez le constater, la fonction Posts est assez longue. Il faut (de manière simplifiée) :

- Créer notre tableau vide avec useState
- Écrire la fonction asynchrone
- Récupérer la réponse de l'API
- Stocker la réponse
- Utiliser le setPosts pour définir le tableau de la réponse...

Le processus prend du temps, de l'énergie et peut être répétitif si vous devez l'effectuer plusieurs fois. Si vous n'avez à utiliser cette requête qu'une seule fois, la création d'un Hook personnalisé ne se révélera pas pertinent. Ici, nous souhaitons récupérer les Posts, les Commentaires et les Utilisateurs.

Nous allons donc isoler la logique de récupération de données au sein d'une fonction qui sera notre Hook.

```

1 function useFetch(url) {
2   const [state, setState] = useState({
3     items: []
4   })
5
6   useEffect(function () {
7     (async function () {
8       const response = await fetch(url)
9       const responseData = await response.json()
10      if(response.ok) {
11        setState({
12          items: responseData
13        })
14      } else {
15        alert(JSON.stringify(responseData))
16      }
17    }) ()
18  }, [])
19
20  return [
21    state.items
22  ]
23 }

```

Dans notre fonction useFetch, nous passons un paramètre que nous appellerons url, ce sera, comme son nom l'indique, l'url à laquelle nous souhaitons récupérer les données. Nous créons notre état avec useState et nous lui assignons par défaut un objet qui contiendra nos items qui est un tableau vide. (Pour le moment)

Nous écrivons normalement notre fetch au sein d'un useEffect en passant la propriété url en paramètre du fetch. Si la réponse est "ok", alors nous utilisons setState pour que le tableau items contienne la réponse définie au-dessus. Pour finir, nous retournons notre tableau items. Cela va rendre la tâche plus facile et va permettre d'optimiser nos requêtes. Maintenant, pour récupérer quelque chose d'une API, nous n'aurons plus qu'à utiliser la fonction useFetch.

Exemple d'utilisation :

```

1 import './App.css';
2 import React, { useEffect, useState } from 'react';
3 import { FlatList, StyleSheet, Text, View } from 'react-native';
4
5 function TodoList() {
6   const [items] = useFetch('https://jsonplaceholder.typicode.com/todos?_limit=10')
7
8   return <FlatList>
9     {items.map(item => <Text>{item.title}</Text>)}
10  </FlatList>
11 }
12
13 function Comments() {
14   const [ items ] = useFetch('https://jsonplaceholder.typicode.com/comments?_limit=10')
15
16   return <FlatList>
17     {items.map(item => <Text>{item.body}</Text>)}
18  </FlatList>
19 }
20
21 function App() {
22
23   return (
24     <View style={styles.app}>
25       <TodoList />
26       <Comments />
27     </View>
28   );
29 }
30
31 export default App;

```

Revenons-en au cours...

Insérer des données

Maintenant, dans notre composant React Native, on peut ouvrir la base de données et récupérer les données.

Pour cela, on utilise un premier hook `useEffect()` qui va créer les tables au lancement, et un autre qui va récupérer les données. On crée également un bouton qui permet d'ajouter un produit aléatoire, suite à quoi on déclenche une nouvelle requête de récupération des données, grâce à un hook personnalisé nommé `useForceUpdate`, qui permet de forcer un rendu graphique à la demande.

Exemple

```

1 import React, { useState, useEffect } from "react";
2 import { View, Button, Text, StyleSheet } from "react-native";
3 import Constants from "expo-constants";
4 import * as SQLite from "expo-sqlite";
5
6 const db = SQLite.openDatabase("test.db");
7
8 export default function App() {
9   const [tables, setTables] = React.useState(null);
10   const [forcedUpdateId, forceUpdate] = useForceUpdate();
11
12   // Ce code permet d'initialiser les tables au lancement de l'application

```

```

13  useEffect(() => {
14    db.exec([{ sql: "PRAGMA foreign_keys = ON;", args: [] }], false, () =>
15      console.log("Foreign keys turned on")
16    });
17
18    db.transaction((tx) => {
19      tx.executeSql(
20        "CREATE TABLE IF NOT EXISTS Users (id INTEGER PRIMARY KEY, first_name TEXT NOT NULL,
21        last_name TEXT NOT NULL, email TEXT NOT NULL UNIQUE);",
22        [],
23        () => console.log("Table Users created"),
24        (e) => console.warn("Table Users error", e)
25      );
26
27      tx.executeSql(
28        "CREATE TABLE IF NOT EXISTS Products (id INTEGER PRIMARY KEY, title TEXT NOT NULL,
29        description TEXT NOT NULL, quantity INTEGER);",
30        [],
31        () => console.log("Table Products created"),
32        (e) => console.warn("Table Products error", e)
33      );
34
35      tx.executeSql(
36        "CREATE TABLE IF NOT EXISTS WishLists (id INTEGER PRIMARY KEY, owner_id INTEGER, title
37        TEXT NOT NULL, creation_date INTEGER NOT NULL, FOREIGN KEY (owner_id) REFERENCES Users (id) ON
38        DELETE CASCADE);",
39        [],
40        () => console.log("Table WishLists created"),
41        (e) => console.warn("Table WishLists error", e)
42      );
43
44      tx.executeSql(
45        "CREATE TABLE IF NOT EXISTS WhishList_Products (whishlist_id INTEGER, product_id
46        INTEGER, list_order INTEGER, PRIMARY KEY (whishlist_id, product_id, list_order), FOREIGN KEY
47        (whishlist_id) REFERENCES WishLists (id) ON DELETE CASCADE, FOREIGN KEY (product_id)
48        REFERENCES Products (id) ON DELETE CASCADE);",
49        [],
50        () => console.log("Table WhishList_Products created"),
51        (e) => console.warn("Table WhishList_Products error", e)
52      );
53    });
54  }, []);
55
56  // Récupère via une requête SQL les données quand forceUpdatedId change (déclenché
57  manuellement)
58  useEffect(() => {
59    db.transaction((tx) => {
60      tx.executeSql(
61        "SELECT * FROM Products",
62        [],
63        (_, result) => {
64          setTables(result.rows._array);
65        },
66        console.warn
67      );
68    });
69  }, [forcedUpdateId]);
70
71  return (
72    <View style={styles.container}>
73      <Button

```

```

65     title="Ajouter un objet aléatoire"
66     onPress={() => {
67         db.transaction((t) => {
68             t.executeSql(
69                 "INSERT INTO Products (title, description, quantity) VALUES (?, ?, 100);",
70                 [
71                     "Produit" + Math.round(Math.random() * 100),
72                     "Description de l'article",
73                 ],
74                 () => {
75                     // Si tout se passe bien, on utilise forceUpdate qui va incrémenter un
compteur et donc déclencher notre hook qui va refaire la requete SQL
76                     forceUpdate();
77                 },
78                 (e) => console.warn("Error on product insertion", e)
79             );
80         });
81     }}
82     />
83     <Text>{JSON.stringify(tables, null, 2)}</Text>
84 </View>
85 );
86 }
87
88 const styles = StyleSheet.create({
89     container: {
90         flex: 1,
91         paddingTop: Constants.statusBarHeight,
92     },
93 });
94
95 // Fonction utilitaire qui permet de forcer un re-rendu.
96 const useForceUpdate = () => {
97     const [state, setState] = useState(false);
98     return [state, () => setState(prev => prev+1)];
99 };

```

2:12



Ajouter un objet aléatoire

```
[
  {
    "id": 1,
    "title": "Jambon50",
    "description": "Description de l'article",
    "quantity": 100
  },
  {
    "id": 2,
    "title": "Jambon27",
    "description": "Description de l'article",
    "quantity": 100
  },
  {
    "id": 3,
    "title": "Produit70",
    "description": "Description de l'article",
    "quantity": 100
  },
  {
    "id": 4,
    "title": "Produit20",
    "description": "Description de l'article",
    "quantity": 100
  },
  {
    "id": 5,
    "title": "Produit38",
    "description": "Description de l'article",
    "quantity": 100
  },
  {
    "id": 6,
    "title": "Produit52",
    "description": "Description de l'article",
    "quantity": 100
  },
  {
    "id": 7,
    "title": "Produit53",
    "description": "Description de l'article",
    "quantity": 100
  },
  {
    "id": 8,
    "title": "Produit13",
    "description": "Description de l'article",
    "quantity": 100
  }
]
```

La plus grosse partie de notre code va donc être exécutée à chaque fois dans une transaction. Pour cela, on utilise l'instruction `db.transaction` qui prends une `callback` en paramètre, en lui passant un paramètre que l'on nomme ici `tx` (pour transaction). `tx` est l'instance délimitée de la BDD pour cette transaction, les commandes au sein de la `callback` l'utilisent donc.

Dans une transaction, on utilise l'instruction `tx.executeSql` qui prend en argument une commande SQL en premier paramètre, un tableau qui sert à remplacer d'éventuels « ? » dans cette requête (cela permet de sécuriser les paramètres contre des attaques par injection SQL et de faciliter la saisie des arguments). Puis, enfin, en troisième paramètre, une `callback` de succès et, en quatrième, une `callback` d'échec.

Intégrer avec les données

Pour récupérer des données, on utilise l'instruction SQL `SELECT * from <nom_de_la_table>;` et on peut par la suite récupérer une propriété `_array`, qui est un tableau permettant d'accéder aux données issues de la requête.

```
1 Select * from Products;
```

Pour l'insertion, on utilise l'instruction `INSERT INTO <table> (field1, field2, ...) VALUES (value1, value2, ...);`

```
1 INSERT INTO Products (title, description, quantity) VALUES ('Produit 1', 'Une description',
100);
```

Pour la mise à jour d'un enregistrement, il faut utiliser la syntaxe `UPDATE <table> SET <column> = <new_value> WHERE <condition>;`.

Par exemple, pour mettre à jour le premier produit :

```
1 UPDATE Products SET title = 'Super Jambon 50' WHERE id = 1;
```

Pour la suppression, il faut utiliser la syntaxe `DELETE FROM <table> WHERE <condition>;`. Pour supprimer le produit 1, cette fois, il faudrait écrire :

```
1 DELETE FROM Products where id = 1;
```

Ces commandes SQL sont toujours à utiliser via l'instruction `tx.executeSql` et sont propres au langage SQL. Pour plus d'informations, il est possible de prototyper sur le site <https://sqliteonline.com/> et d'en apprendre plus ici : <https://www.sqlitetutorial.net/>.

Syntaxe À retenir

- Expo permet d'utiliser SQLite dans une application React Native et supporte la majorité des opérations classiques SQL.
- Il devient alors possible de stocker des valeurs et de les récupérer grâce à une API JavaScript, qui communique avec la base de données.
- Pour chaque instruction effectuée, il est possible de passer une `callback` différente en cas de succès ou en cas d'erreur.

Complément

- <https://sqliteonline.com/>
- <https://www.sqlitetutorial.net/>

V. Exercice : Appliquer la notion

Question

[solution n°2 p.22]

Améliorez l'exemple vu précédemment pour supprimer un objet de la liste via un bouton, en utilisant un champ de saisie qui prend en paramètre l'identifiant de l'objet à supprimer.

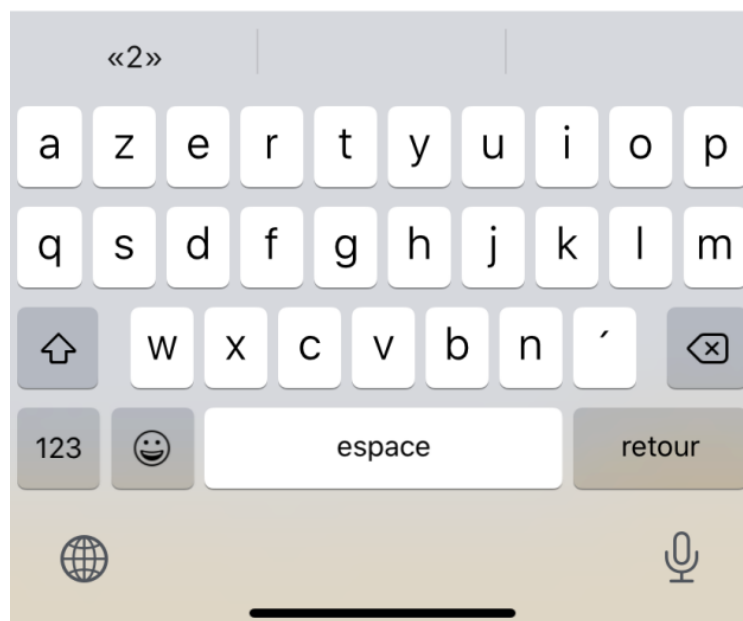
Le code de l'exemple est disponible en indice et le rendu final escompté ci-dessous.

5:12 📶 🔋

Ajouter un objet aléatoire

Supprimer un objet

```
[
  {
    "id": 1,
    "title": "Produit1",
    "description": "Description de l'article",
    "quantity": 100
  },
  {
    "id": 2,
    "title": "Produit5",
    "description": "Description de l'article",
    "quantity": 100
  },
  {
    "id": 3,
    "title": "Produit36",
    "description": "Description de l'article",
    "quantity": 100
  }
]
```



Indice :

```

1 import React, { useState, useEffect } from "react";
2 import { View, Button, Text, StyleSheet } from "react-native";
3 import Constants from "expo-constants";
4 import * as SQLite from "expo-sqlite";
5
6 const db = SQLite.openDatabase("test.db");
7
8 export default function App() {
9   const [tables, setTables] = React.useState(null);
10   const [forcedUpdateId, forceUpdate] = useForceUpdate();
11
12   // Ce code permet d'initialiser les tables au lancement de l'application
13   useEffect(() => {
14     db.exec([{ sql: "PRAGMA foreign_keys = ON;", args: [] }], false, () =>
15       console.log("Foreign keys turned on")
16     );
17
18     db.transaction((tx) => {
19       tx.executeSql(
20         "CREATE TABLE IF NOT EXISTS Users (id INTEGER PRIMARY KEY, first_name TEXT NOT NULL,
21         last_name TEXT NOT NULL, email TEXT NOT NULL UNIQUE);",
22         [],
23         () => console.log("Table Users created"),
24         (e) => console.warn("Table Users error", e)
25       );
26
27       tx.executeSql(
28         "CREATE TABLE IF NOT EXISTS Products (id INTEGER PRIMARY KEY, title TEXT NOT NULL,
29         description TEXT NOT NULL, quantity INTEGER);",
30         [],
31         () => console.log("Table Products created"),
32         (e) => console.warn("Table Products error", e)
33       );
34
35       tx.executeSql(
36         "CREATE TABLE IF NOT EXISTS WishLists (id INTEGER PRIMARY KEY, owner_id INTEGER, title
37         TEXT NOT NULL, creation_date INTEGER NOT NULL, FOREIGN KEY (owner_id) REFERENCES Users (id) ON
38         DELETE CASCADE);",
39         [],
40         () => console.log("Table WishLists created"),
41         (e) => console.warn("Table WishLists error", e)
42       );
43
44       tx.executeSql(
45         "CREATE TABLE IF NOT EXISTS WhishList_Products (whishlist_id INTEGER, product_id
46         INTEGER, list_order INTEGER, PRIMARY KEY (whishlist_id, product_id, list_order), FOREIGN KEY
47         (whishlist_id) REFERENCES WishLists (id) ON DELETE CASCADE, FOREIGN KEY (product_id)
48         REFERENCES Products (id) ON DELETE CASCADE);",
49         [],
50         () => console.log("Table WhishList_Products created"),
51         (e) => console.warn("Table WhishList_Products error", e)
52       );
53     });
54   }, []);
55
56   // Récupère via une requête SQL les données quand forceUpdatedId change (déclenché
57   manuellement)
58   useEffect(() => {
59     db.transaction((tx) => {
60       tx.executeSql(
61         "SELECT * FROM Products",
62         [],

```



```

53     (_, result) => {
54         setTables(result.rows._array);
55     },
56     console.warn
57 );
58 });
59 }, [forcedUpdateId]);
60
61
62 return (
63     <View style={styles.container}>
64         <Button
65             title="Ajouter un objet aléatoire"
66             onPress={() => {
67                 db.transaction((t) => {
68                     t.executeSql(
69                         "INSERT INTO Products (title, description, quantity) VALUES (?, ?, 100);",
70                         [
71                             "Produit" + Math.round(Math.random() * 100),
72                             "Description de l'article",
73                         ],
74                     () => {
75                         // Si tout se passe bien, on utilise forceUpdate qui va incrémenter un
compteur et donc déclencher notre hook qui va refaire la requête SQL
76                         forceUpdate();
77                     },
78                     (e) => console.warn("Error on product insertion", e)
79                 );
80             });
81         }
82     />
83     <Text>{JSON.stringify(tables, null, 2)}</Text>
84 </View>
85 );
86 }
87
88 const styles = StyleSheet.create({
89     container: {
90         flex: 1,
91         paddingTop: Constants.statusBarHeight,
92     },
93 });
94
95 // Fonction utilitaire qui permet de forcer un re-rendu.
96 const useForceUpdate = () => {
97     const [state, setState] = useState(false);
98     return [state, () => setState(prev => prev+1)];
99 };

```

VI. Essentiel

SQLite est un moteur de base de données relationnelle embarqué dans un fichier. Il permet à notre application, en utilisant Expo, de persister entre les différents redémarrages.

Il utilise la syntaxe SQL, comparable à ce qu'il se fait dans le développement back-end, et qui permet donc des requêtes assez avancées pour les données. Ces requêtes s'effectuent dans des transactions afin d'avoir toujours une intégrité de données parfaite.

VII. Auto-évaluation

A. Exercice final

Exercice 1

[solution n°3 p.24]

Exercice

SQLite est...

- ☐ Une base de données NoSQL
- ☐ Une base de données relationnelle

Exercice

Pour utiliser SQLite sur React Native avec Expo, j'utilise la commande...

- ☐ expo install expo-sqlite
- ☐ expo add sqlite
- ☐ expo bootstrap expo-sqlite

Exercice

Qu'est-ce qu'une clé primaire ?

- ☐ C'est une contrainte que l'on applique à une colonne permettant de trier les enregistrements
- ☐ C'est une référence vers un champ d'une autre table
- ☐ C'est une contrainte que l'on applique sur un enregistrement, qui doit permettre de l'identifier et qui doit être unique dans la table

Exercice

Qu'est-ce qu'une clé étrangère ?

- ☐ C'est une contrainte que l'on applique à une colonne permettant de trier les enregistrements
- ☐ C'est une contrainte permettant d'indiquer une dépendance vis-à-vis d'un autre enregistrement dans une autre table et de définir les comportements à adopter en cas de modification ou de suppression de cet enregistrement
- ☐ C'est un *addon* permettant de traduire dans une autre langue la valeur d'un champ

Exercice

Une transaction permet...

- ☐ D'optimiser les performances d'accès à la base de données
- ☐ De factoriser le code des requêtes pour qu'il soit plus lisible et performant
- ☐ De garantir que toutes les opérations en son sein vont être exécutées dans leur ensemble, et avec les bonnes valeurs de référence

Exercice

Pour ne pas faire planter le système, si une table existe déjà, j'utilise la syntaxe...

- ☐ CREATE TABLE ONLY NEW
- ☐ CREATE TABLE IF NOT EXISTS
- ☐ FIND OR CREATE TABLE

Exercice

À quoi cela sert d'utiliser un ? dans une requête SQL ?

- ☐ Pouvoir facilement passer des arguments à la requête sans modifier une chaîne de caractères
- ☐ Sécuriser la requête SQL contre d'éventuelles tentatives d'injection SQL
- ☐ Mettre le champ à `null`, car on ne sait pas la valeur

Exercice

Si je désinstalle l'application...

- ☐ La base de données est vidée
- ☐ La base de données n'est pas vidée et les données seront là si je réinstalle

Exercice

Pour gérer une date avec SQLite...

- ☐ J'utilise un entier et je stocke un timestamp
- ☐ J'utilise le type `Date` de SQLITE

Exercice

Parmi ces types de données, quels sont ceux qui existent en SQLite ?

- ☐ INTEGER
- ☐ DATE
- ☐ NULL
- ☐ VARCHAR

B. Exercice : Défi

En réutilisant le modèle de données SQL vu dans ce cours, réaliser une interface qui liste des produits depuis la base de données et qui affiche en bas de page une liste de souhaits avec des produits à l'intérieur.

Question

[solution n°4 p.27]

Chaque produit peut être ajouté ou enlevé grâce à un bouton dédié à côté de l'élément.

Si je relance l'application mon état de liste est conservé.

Rendu escompté en exemple ci-dessous.

[cf. sql3.mp4]

Solutions des exercices

p.6 Solution n°1

En se basant sur la demande, on peut définir que deux tables seront nécessaires : `Users` et `Recipes`. Comme les utilisateurs peuvent voter pour des recettes, on peut modéliser une table référençant les utilisateurs et les recettes, et permettant de stocker la note d'un utilisateur pour une recette, `Recipes_Notations` :

```

1 // Table Users
2 | id | first_name | last_name | email |
3 |----|-----|-----|-----|
4 | 1 | andreas   | Hanss    | contact@codingspark.io |
5 | 2 | hugo      | Delatour  | hugo@jymiz.com |
6
7 // Table Recipes
8 | id | title | description |
9 |----|-----|-----|
10 | 1 | Gratin de courgettes | Mettre l'oignon et l'ail dans le mixeur et hacher 5
    secondes vitesse 5. Laver les courgettes et les couper en 2 dans le sens de la longueur puis
    les couper en (demi) rondelles. Cuire pendant 15 minutes. |
11 | 2 | Gratin de pommes de terre | Epluchez et coupez les pommes de terre en fines lamelles
    d'environ 2/3 mm. Déposez-les dans un plat à gratin, versez la crème. Salez et poivrez.
    Parsemez de fromage et enfournez pour 30 min. |
12
13 // Table Recipes_Notations
14 | user_id | recipe_id | note |
15 |-----|-----|-----|
16 | 1 | 1 | 5 |
17 | 2 | 1 | 4 |
18 | 1 | 2 | 3 |

```

Lors de l'écriture du script SQL permettant la création des tables et afin d'assurer l'identification d'un utilisateur, on peut spécifier que plusieurs utilisateurs ne peuvent pas avoir le même mail grâce à la contrainte `UNIQUE` :

```

1 // Script SQL
2 CREATE TABLE Users (
3   id INTEGER PRIMARY KEY,
4   first_name TEXT NOT NULL,
5   last_name TEXT NOT NULL,
6   email TEXT NOT NULL UNIQUE
7 );
8 CREATE TABLE Recipes (
9   id INTEGER PRIMARY KEY,
10  title TEXT NOT NULL,
11  description TEXT NOT NULL
12 );
13 CREATE TABLE Recipes_Notations (
14   user_id INTEGER,
15   recipe_id INTEGER,
16   note INTEGER NOT NULL,
17   PRIMARY KEY (user_id, recipe_id),
18   FOREIGN KEY (user_id) REFERENCES Users (id)
19     ON DELETE CASCADE
20   FOREIGN KEY (recipe_id) REFERENCES Recipes (id)
21     ON DELETE CASCADE
22 );

```

Il est important ici de remarquer comment se définit une clé primaire composée de plusieurs champs, avec la contrainte `PRIMARY KEY (user_id, recipe_id)`.

p. 15 Solution n°2

```

1 import React, { useState, useEffect } from "react";
2 import { View, TextInput, Button, Text, StyleSheet } from "react-native";
3 import Constants from "expo-constants";
4 import * as SQLite from "expo-sqlite";
5
6 const db = SQLite.openDatabase("test.db");
7 export default function App() {
8   const [tables, setTables] = React.useState(null);
9   const [forcedUpdateId, forceUpdate] = useForceUpdate();
10  const [toDeleteId, setToDeleteId] = useState();
11
12  // Ce code permet d'initialiser les tables au lancement de l'application
13  useEffect(() => {
14    db.exec([ { sql: "PRAGMA foreign_keys = ON;", args: [] } ], false, () =>
15      console.log("Foreign keys turned on")
16    );
17
18    db.transaction((t) => {
19      t.executeSql(
20        "CREATE TABLE IF NOT EXISTS Users (id INTEGER PRIMARY KEY, first_name TEXT NOT NULL,
21        last_name TEXT NOT NULL, email TEXT NOT NULL UNIQUE);",
22        [],
23        () => console.log("Table Users created"),
24        (e) => console.warn("Table Users error", e)
25      );
26
27      t.executeSql(
28        "CREATE TABLE IF NOT EXISTS Products (id INTEGER PRIMARY KEY, title TEXT NOT NULL,
29        description TEXT NOT NULL, quantity INTEGER);",
30        [],
31        () => console.log("Table Products created"),
32        (e) => console.warn("Table Products error", e)
33      );
34
35      t.executeSql(
36        "CREATE TABLE IF NOT EXISTS WishLists (id INTEGER PRIMARY KEY, owner_id INTEGER,
37        title TEXT NOT NULL, creation_date INTEGER NOT NULL, FOREIGN KEY (owner_id) REFERENCES Users
38        (id) ON DELETE CASCADE);",
39        [],
40        () => console.log("Table WishLists created"),
41        (e) => console.warn("Table WishLists error", e)
42      );
43
44      t.executeSql(
45        "CREATE TABLE IF NOT EXISTS WhishList_Products (whishlist_id INTEGER, product_id
46        INTEGER, list_order INTEGER, PRIMARY KEY (whishlist_id, product_id, list_order), FOREIGN KEY
47        (whishlist_id) REFERENCES WishLists (id) ON DELETE CASCADE, FOREIGN KEY (product_id)
48        REFERENCES Products (id) ON DELETE CASCADE);",
49        [],
50        () => console.log("Table WhishList_Products created"),
51        (e) => console.warn("Table WhishList_Products error", e)
52      );
53    });
54  }, []);
55
56  // Récupère via une requête SQL les données quand forceUpdatedId change (déclenché
57  manuellement)
58  useEffect(() => {
59    db.transaction((tx) => {
60      tx.executeSql(
61        "select * from Products",
62

```

```

52     [] ,
53     (_, result) => {
54         setTables(result.rows._array);
55     },
56     console.warn
57 );
58 });
59 }, [forcedUpdateId]);
60
61
62 return (
63     <View style={styles.container}>
64         <Button
65             title="Ajouter un objet aléatoire"
66             onPress={() => {
67                 db.transaction((t) => {
68                     t.executeSql(
69                         "INSERT INTO Products (title, description, quantity) VALUES (?, ?, 100);",
70                         [
71                             "Produit" + Math.round(Math.random() * 100),
72                             "Description de l'article",
73                         ],
74                         () => {
75                             // Si tout se passe bien, on utilise forceUpdate qui va incrémenter un
compteur et donc déclencher notre hook qui va refaire la requete SQL
76                             forceUpdate();
77                         },
78                         (e) => console.warn("Error on product insertion", e)
79                     );
80                 });
81             }}
82         />
83         <TextInput
84             style={{ borderWidth: 1 }}
85             placeholder="Renseigner l'identifiant à supprimer"
86             onChangeText={(text) => {
87                 setToDeleteId(Number(text));
88             }}
89             value={toDeleteId}
90         />
91         <Button
92             title="Supprimer un objet"
93             onPress={() => {
94                 db.transaction((t) => {
95                     t.executeSql(
96                         "DELETE FROM Products WHERE id = ?;",
97                         [toDeleteId],
98                         () => {
99                             // Si tout se passe bien, on utilise forceUpdate qui va incrémenter un
compteur et donc déclencher notre hook qui va refaire la requete SQL
100                             forceUpdate();
101                         }
102                     );
103                 });
104             }}
105         />
106         <Text>{JSON.stringify(tables, null, 2)}</Text>
107     </View>
108 );

```

```

109 }
110
111 const styles = StyleSheet.create({
112   container: {
113     flex: 1,
114     paddingTop: Constants.statusBarHeight,
115   },
116 });
117
118 // Fonction utilitaire qui permet de forcer un re-rendu.
119 const useForceUpdate = () => {
120   const [state, setState] = useState(false);
121   return [state, () => setState(prev => prev+1)];
122 };

```

Pour réaliser cela, il suffit d'ajouter un champ de saisie, ainsi qu'une variable dans le *state* stockant sa valeur :

```

1 [...]
2 const [toDeleteId, setToDeleteId] = useState();
3 [...]
4 return (
5 [...]
6   <TextInput
7     style={{ borderWidth: 1 }}
8     placeholder="Renseigner l'identifiant à supprimer"
9     onChangeText={(text) => {
10       setToDeleteId(Number(text));
11     }}
12     value={toDeleteId}
13   />
14 [...]
15 )

```

Puis on ajoute un bouton qui va exécuter une requête SQL de suppression en prenant la valeur contenue dans le *state* en paramètre :

```


1 <Button
2   title="Supprimer un objet"
3   onPress={() => {
4     db.transaction((t) => {
5       t.executeSql(
6         "DELETE FROM Products WHERE id = ?;",
7         [toDeleteId],
8         () => {
9           // Si tout se passe bien, on utilise forceUpdate qui va incrémenter un compteur et
10             donc déclencher notre hook qui va refaire la requete SQL
11             forceUpdate();
12         }
13       );
14     });
15   />

```

Exercice p. 18 Solution n°3


Exercice

SQLite est...

- ☐ Une base de données NoSQL
- ☒ Une base de données relationnelle
-  C'est une base de données relationnelle embarquée avec une structure en table, typée et figée.


Exercice

Pour utiliser SQLite sur React Native avec Expo, j'utilise la commande...

- ☒ expo install expo-sqlite
- ☐ expo add sqlite
- ☐ expo bootstrap expo-sqlite
-  La seule bonne commande est la première, les autres n'existent pas.


Exercice

Qu'est-ce qu'une clé primaire ?

- ☐ C'est une contrainte que l'on applique à une colonne permettant de trier les enregistrements
- ☐ C'est une référence vers un champ d'une autre table
- ☒ C'est une contrainte que l'on applique sur un enregistrement, qui doit permettre de l'identifier et qui doit être unique dans la table
-  Une clé primaire peut être simple ou composée de plusieurs champs, elle permet d'identifier un enregistrement et s'assure de son unicité dans la table.

Exercice

Qu'est-ce qu'une clé étrangère ?

- ☐ C'est une contrainte que l'on applique à une colonne permettant de trier les enregistrements
- ☒ C'est une contrainte permettant d'indiquer une dépendance vis-à-vis d'un autre enregistrement dans une autre table et de définir les comportements à adopter en cas de modification ou de suppression de cet enregistrement
- ☐ C'est un *addon* permettant de traduire dans une autre langue la valeur d'un champ
-  Une clé étrangère est une référence vers un champ permettant d'identifier un enregistrement dans une autre table, généralement la clé primaire de cet enregistrement.

Exercice

Une transaction permet...

- ☐ D'optimiser les performances d'accès à la base de données
- ☐ De factoriser le code des requêtes pour qu'il soit plus lisible et performant
- ☒ De garantir que toutes les opérations en son sein vont être exécutées dans leur ensemble, et avec les bonnes valeurs de référence

- Q La transaction permet de sécuriser les modifications de plusieurs enregistrements différents et de garantir que ces dernières soient toutes exécutées d'une traite.

Exercice

Pour ne pas faire planter le système, si une table existe déjà, j'utilise la syntaxe...

- ☐ CREATE TABLE ONLY NEW
- ☒ CREATE TABLE IF NOT EXISTS
- ☐ FIND OR CREATE TABLE

- Q Cette instruction permet de s'assurer que l'instruction `CREATE TABLE` ne sera exécutée que si celle-ci n'est pas déjà présente dans le système.

Exercice

À quoi cela sert d'utiliser un `?` dans une requête SQL ?

- ☒ Pouvoir facilement passer des arguments à la requête sans modifier une chaîne de caractères
- ☒ Sécuriser la requête SQL contre d'éventuelles tentatives d'injection SQL
- ☐ Mettre le champ à `null`, car on ne sait pas la valeur

- Q Le `?` dans l'écriture d'une requête SQL permet de lui passer des paramètres sans faire de concaténation de chaînes. En effet, ils seront automatiquement remplacés par les valeurs passées en paramètres. De plus, cette méthode permet de se protéger des injections SQL, car des vérifications des valeurs fournies sont réalisées par la librairie au moment du remplacement.

Exercice

Si je désinstalle l'application...

- ☒ La base de données est vidée
- ☐ La base de données n'est pas vidée et les données seront là si je réinstalle

- Q La base de données se vide, car l'accès au système de fichiers de l'application est nettoyé au moment où l'application est désinstallée.

Exercice

Pour gérer une date avec SQLite...


- ☒ J'utilise un entier et je stocke un timestamp
- ☐ J'utilise le type `Date` de SQLite

- Q Le type `Date` de SQLite n'existe malheureusement pas, le plus simple est donc de stocker un entier qui contient un timestamp UNIX.

Exercice

Parmi ces types de données, quels sont ceux qui existent en SQLite ?

- ☒ INTEGER
- ☐ DATE
- ☒ NULL
- ☐ VARCHAR

 Le type VARCHAR est remplacé par TEXT en SQLite, et le type Date n'existe malheureusement pas.

p. 19 Solution n°4

```

1 import React, { useState, useEffect } from "react";
2 import { View, Button, Text, StyleSheet } from "react-native";
3 import Constants from "expo-constants";
4 import * as SQLite from "expo-sqlite";
5 import { ScrollView } from "react-native-gesture-handler";
6
7 const db = SQLite.openDatabase("test.db");
8
9 export default function App() {
10   const [productsList, setProductsList] = React.useState([]);
11   const [wishlist, setWhishList] = React.useState();
12   const [elementsInWishlist, setElementsInWishlist] = React.useState([]);
13   const [forcedUpdateId, forceUpdate] = useForceUpdate();
14
15   // Ce code permet d'initialiser les tables au lancement de l'application
16   useEffect(() => {
17     // Active le mode clé étrangère
18     db.exec([ { sql: "PRAGMA foreign_keys = ON;", args: [] } ], false, () =>
19       console.log("Foreign keys turned on")
20     );
21
22     db.transaction(tableCreationTransaction);
23   }, []);
24
25   // Récupère via une requête SQL les données quand forceUpdatedId change (déclenché
26   // manuellement)
27   useEffect(() => {
28     db.transaction((tx) => {
29       // Récupère la liste des produits
30       tx.executeSql("select * from Products", [], (_, result) => {
31         setProductsList(result.rows._array);
32       });
33
34       // Récupère la liste
35       tx.executeSql("select * from WishLists", [], (_, result) => {
36         setWhishList(result.rows._array.pop()); // Get first one
37       });
38
39       if (wishlist) {
40         // Récupère la liste des éléments dans la liste
41         tx.executeSql(
42           "select * from WhishList_Products W WHERE whishlist_id = ?;",
43           [wishlist.id],
44           (_, result) => {
45             setElementsInWishlist(result.rows._array);
46           }
47         );
48       }
49     });
50   });
51 }

```

```

45     }
46   );
47 }
48 });
49 }, [forcedUpdateId, wishList]);
50
51
52 return (
53   <View style={styles.container}>
54     <Button
55       disabled={!isNil(wishList)}
56       title={isNil(wishList) ? "Cliquez pour initialiser" : "Déjà initialisé"}
57       onPress={() => {
58         db.transaction((t) => {
59           // On créé 3 produits
60           [1, 2, 3].forEach(
61             (el) =>
62               t.executeSql(
63                 "INSERT INTO Products (title, description, quantity) VALUES (?, ?, ?);",
64                 ["Produit" + el, "Description de l'article", 100]
65               ),
66             () => console.log("Products Created"),
67             (e) => console.log("Error on products creation", e)
68           );
69
70           // On créé l'utilisateur
71           t.executeSql(
72             "INSERT INTO Users (first_name, last_name, email) VALUES (?, ?, ?);",
73             ["Andréas", "HANSS", "contact@codingspark.io"],
74             (result) => {
75               console.log("Users Created");
76               // Create WhishList for user
77               t.executeSql(
78                 "INSERT INTO WishLists (owner_id, title, creation_date) VALUES (?, ?, ?);",
79                 [1, "La liste d'andréas", Date.now()], // On met 1 en dur car on sait que
80                 c'est le premier dans notre cas, sinon il faudrait récupérer la valeur de l'ID
81                 () => console.log("Whishlist created"),
82                 (e) => console.log("Error on wishlist creation", e)
83               );
84             },
85             (e) => console.log("Error on Users creation", e)
86           );
87
88           // Si tout se passe bien, on utilise forceUpdate qui va incrémenter un compteur
89           // et donc déclencher notre hook qui va refaire la requete SQL
90           forceUpdate();
91         });
92       }
93     />
94     <ScrollView contentContainerStyle={{ padding: 20 }}>
95       {productsList.map((product) => (
96         <ObjectLine
97           key={product.id}
98           title={product.title}
99           listId={wishList ? wishList.id : undefined}
100           id={product.id}
101           isInBucket={elementsInWishlist.find(
102             (el) => el.product_id === product.id
103           )}
104         )}
105       )}
106     </ScrollView>
107   </View>
108 );

```

```

102         forceUpdate={forceUpdate}
103         description={product.description}
104     />
105     )}}
106 </ScrollView>
107 {!isNil(wishList) && (
108     <View style={styles.bottomContainer}>
109         <Text>Ma liste : {wishList.title}</Text>
110         <Text>{JSON.stringify(elementsInWishlist)}</Text>
111     </View>
112 )}
113 </View>
114 );
115 }
116
117 const ObjectLine = (props) => {
118     return (
119         <View style={styles.objectLineStyles}>
120             <View>
121                 <Text>{props.title}</Text>
122                 <Text>{props.description}</Text>
123             </View>
124             <View>
125                 <Button
126                     disabled={isNil(props.listId)}
127                     title={props.isInBucket ? "Enlever" : "Ajouter"}
128                     onPress={() => {
129                         db.transaction((tx) => {
130                             // Si l'élément est déjà dans la liste on l'enlève sinon on l'ajoute. Dans tous
131                             les cas on déclenche un re-rendu
132                             props.isInBucket
133                                 ? tx.executeSql(
134                                     "DELETE FROM WhishList_Products WHERE whishlist_id = ? AND product_id =
135                                     ?;",
136                                     [props.listId, props.id],
137                                     () => {
138                                         props.forceUpdate();
139                                     }
140                                 )
141                                 : tx.executeSql(
142                                     "INSERT INTO WhishList_Products (whishlist_id, product_id, list_order)
143                                     VALUES (?, ?, ?);",
144                                     [props.listId, props.id, randomIntFromInterval(1, 1000)],
145                                     () => {
146                                         props.forceUpdate();
147                                     }
148                                 );
149                             }
150                         );
151                     }
152                 </Button>
153             </View>
154         </View>
155     );
156 }
157
158 const styles = StyleSheet.create({
159     container: {
160         flex: 1,
161         paddingTop: Constants.statusBarHeight,

```

```

158   },
159   bottomContainer: {
160     height: 100,
161     backgroundColor: "lightgrey",
162     paddingBottom: Constants.statusBarHeight,
163   },
164   objectLineStyles: {
165     flexDirection: "row",
166     justifyContent: "space-between",
167     marginBottom: 20,
168   },
169 });
170
171 // Fonction utilitaire qui permet de forcer un re-rendu.
172 const useForceUpdate = () => {
173   const [state, setState] = useState(false);
174   return [state, () => setState(prev => prev+1)];
175 };
176
177 const isNil = (value) => typeof value === "undefined" || value === null
178
179 const randomIntFromInterval = (min, max) => { // min and max included
180   return Math.floor(Math.random() * (max - min + 1) + min);
181 }
182
183 const tableCreationTransaction =(t) => {
184   t.executeSql(
185     "CREATE TABLE IF NOT EXISTS Users (id INTEGER PRIMARY KEY, first_name TEXT NOT NULL, last_name TEXT NOT NULL, email TEXT NOT NULL UNIQUE);",
186     [],
187     () => console.log("Table Users created"),
188     (e) => console.warn("Table Users error", e)
189   );
190
191   t.executeSql(
192     "CREATE TABLE IF NOT EXISTS Products (id INTEGER PRIMARY KEY, title TEXT NOT NULL, description TEXT NOT NULL, quantity INTEGER);",
193     [],
194     () => console.log("Table Products created"),
195     (e) => console.warn("Table Products error", e)
196   );
197   t.executeSql(
198     "CREATE TABLE IF NOT EXISTS WishLists (id INTEGER PRIMARY KEY, owner_id INTEGER, title TEXT NOT NULL, creation_date INTEGER NOT NULL, FOREIGN KEY (owner_id) REFERENCES Users (id) ON DELETE CASCADE);",
199     [],
200     () => console.log("Table WishLists created"),
201     (e) => console.warn("Table WishLists error", e)
202   );
203   t.executeSql(
204     "CREATE TABLE IF NOT EXISTS WhishList_Products (whishlist_id INTEGER, product_id INTEGER, list_order INTEGER, PRIMARY KEY (whishlist_id, product_id, list_order), FOREIGN KEY (whishlist_id) REFERENCES WishLists (id) ON DELETE CASCADE, FOREIGN KEY (product_id) REFERENCES Products (id) ON DELETE CASCADE);",
205     [],
206     () => console.log("Table WhishList_Products created"),
207     (e) => console.warn("Table WhishList_Products error", e)
208   );
209 }

```

Afin de répondre à la demande, on met en place un hook `useEffect` d'initialisation de la base de données qui se lance au démarrage de l'application. Pour cela il exécute, sous forme de transaction, les requêtes fournies par la méthode `tableCreationTransaction()` :

```
1 useEffect(() => {
2   // Active le mode clé étrangère
3   db.exec([ { sql: "PRAGMA foreign_keys = ON;", args: [] } ], false, () =>
4     console.log("Foreign keys turned on")
5   );
6
7   db.transaction(tableCreationTransaction);
8 }, []);
```

Puis un second `useEffect`, s'exécutant à chaque modification du compteur `forcedUpdateId` ou de la liste de souhaits, permet de requêter la base de données pour en récupérer toutes les informations nécessaires : la liste des produits, puis la première liste de souhaits et enfin les souhaits associés à cette liste. Chaque ensemble de données est stocké dans le state de l'application :

```
1 useEffect(() => {
2   db.transaction((tx) => {
3     // Récupère la liste des produits
4     tx.executeSql("select * from Products", [], (_, result) => {
5       setProductsList(result.rows._array);
6     });
7
8     // Récupère la liste
9     tx.executeSql("select * from WishLists", [], (_, result) => {
10      setWishList(result.rows._array.pop()); // Get first one
11    });
12
13    if (wishList) {
14      // Récupère la liste des éléments dans la liste
15      tx.executeSql(
16        "select * from WhishList_Products W WHERE whishlist_id = ?;",
17        [wishList.id],
18        (_, result) => {
19          setElementsInWishlist(result.rows._array);
20        }
21      );
22    }
23  });
24 }, [forcedUpdateId, wishList]);
```

On crée ensuite un bouton permettant d'insérer des données de départ si la base est vide, si celle-ci ne l'est pas on affiche "Déjà initialisé" et on désactive le bouton.

Les données de départ contiennent trois produits, un utilisateur et sa liste de souhaits vide :

```
1 <Button
2   disabled={!isNil(wishList)}
3   title={isNil(wishList) ? "Cliquez pour initialiser" : "Déjà initialisé"}
4   onPress={() => {
5     db.transaction((t) => {
6       // On crée 3 produits
7       [1, 2, 3].forEach(
8         (el) =>
9         t.executeSql(
10           "INSERT INTO Products (title, description, quantity) VALUES (?, ?, ?);",
11           ["Produit" + el, "Description de l'article", 100]
12         ),
13       () => console.log("Products Created"),
```

```

14     (e) => console.log("Error on products creation", e)
15   );
16
17   // On créé l'utilisateur
18   t.executeSql(
19     "INSERT INTO Users (first_name, last_name, email) VALUES (?, ?, ?);",
20     ["Andréas", "HANSS", "contact@codingspark.io"],
21     (result) => {
22       console.log("Users Created");
23       // Create WhishList for user
24       t.executeSql(
25         "INSERT INTO WishLists (owner_id, title, creation_date) VALUES (?, ?, ?);",
26         [1, "La liste d'andréas", Date.now()], // On met 1 en dur car on sait que c'est le
27         premier dans notre cas, sinon il faudrait récupérer la valeur de l'ID
28         () => console.log("Whishlist created"),
29         (e) => console.log("Error on wishlist creation", e)
30       );
31     },
32     (e) => console.log("Error on Users creation", e)
33   );
34
35   // Si tout se passe bien, on utilise forceUpdate qui va incrémenter un compteur et donc
36   déclencher notre hook qui va refaire la requete SQL
37   forceUpdate();
38 }
39 }
40 }
41 }
42 }
43 }
44 }
45 }
46 }
47 }
48 }
49 }
50 }
51 }
52 }
53 }
54 }
55 }
56 }
57 }
58 }
59 }
60 }
61 }
62 }
63 }
64 }
65 }
66 }
67 }
68 }
69 }
70 }
71 }
72 }
73 }
74 }
75 }
76 }
77 }
78 }
79 }
80 }
81 }
82 }
83 }
84 }
85 }
86 }
87 }
88 }
89 }
90 }
91 }
92 }
93 }
94 }
95 }
96 }
97 }
98 }
99 }
100 }

```

Pour ce qui est de l'affichage de la liste on peut factoriser le code en considérant que chaque ligne de la liste est un composant `<ObjectLine>` recevant en props les informations représentant un produit et affichant son titre, sa description et un bouton en permettant l'ajout ou la suppression dans la liste de souhaits.

Pour cela, selon si le produit est présent dans la liste de souhait, une requête d'ajout (INSERT) ou de suppression (DELETE) sera exécutée :

```

1 const ObjectLine = (props) => {
2   return (
3     <View style={styles.objectLineStyle}>
4       <View>
5         <Text>{props.title}</Text>
6         <Text>{props.description}</Text>
7       </View>
8       <View>
9         <Button
10           disabled={isNil(props.listId)}
11           title={props.isInBucket ? "Enlever" : "Ajouter"}
12           onPress={() => {
13             db.transaction((tx) => {
14               // Si l'élément est déjà dans la liste on l'enlève sinon on l'ajoute. Dans tous
15               les cas on déclenche un re-rendu
16               props.isInBucket
17               ? tx.executeSql(
18                 "DELETE FROM WhishList_Products WHERE whishlist_id = ? AND product_id =
19                 ?;",
20                 [props.listId, props.id],
21                 () => {
22                   props.forceUpdate();
23                 }
24               )
25             : tx.executeSql(

```



```

24         "INSERT INTO WhishList_Products (whishlist_id, product_id, list_order)
VALUES (?, ?, ?);",
25         [props.listId, props.id, randomIntFromInterval(1, 1000)],
26         () => {
27             props.forceUpdate();
28         }
29     );
30 });
31 }}
32 />
33 </View>
34 </View>
35 );
36 };

```

On affiche ensuite une liste de ces composants pour chaque produit :

```

1 <ScrollView contentContainerStyle={{ padding: 20 }}>
2   {productsList.map((product) => (
3     <ObjectLine
4       key={product.id}
5       title={product.title}
6       listId={wishlist ? wishlist.id : undefined}
7       id={product.id}
8       isInBucket={elementsInWishlist.find(
9         (el) => el.product_id === product.id
10      )}
11       forceUpdate={forceUpdate}
12       description={product.description}
13     />
14   ))}
15 </ScrollView>

```