Mastodon

Application CRUD avec Express et SQlite en 10 étapes

2019-09-11 #javascript #node #sql

Le but de ce projet ultra simple est de développer une application Node JS pour apprendre comment :

- Créer réellement un site web très-très basique avec Express.
- Gérer la mise à jour d'une base de données SQL (SQlite en l'occurence).

Ce billet n'est qu'un tutoriel pour voir comment ça marche et avoir une base de départ pour me former petit à petit à Node et à Express (et sans doute plus tard à Sequelize). Ce n'est absolument pas un guide des bonnes pratiques à suivre pour développer de "vraies" applications. Ce n'est pas non plus un article pour apprendre à programmer ou pour convaincre qui que ce soit d'utiliser Node, Express ou SQL...

Le code JavaScript final est visible dans l'annexe en fin de billet. Le code complet de l'application est disponible sur **GitHub**.

Pour l'instant, il n'y a pas de site de démonstration du projet terminé. Je n'ai pas (encore) trouvé de solution facile pour l'héberger (surtout avec une base de données SQlite). Je ferai peut-être un autre tutoriel le jour où je m'attaquerai à ce problème. Ça a pris du temps, mais c'est fait : Déployer une application sur Glitch en 5 étapes !

Note : J'ai depuis rédigé un deuxième tutoriel qui reprend exactement celui-ci, mais en se connectant à une base de données PostgreSQL à la place : Application CRUD avec Express et PostgreSQL en 10 étapes.

Sommaire

1. Créer un nouveau projet Node
2. Ajouter des modules au projet Node
3. Créer l'application Express
4. Ajouter des vues EJS
5. Utiliser les vues dans Express
6. Premiers pas avec le module SQlite3
7. Modifier une fiche
8. Créer une nouvelle fiche
9. Supprimer une fiche
10. Conclusion

1. Créer un nouveau projet Node

Créer un dossier pour le projet

On peut commencer au niveau de la ligne de commande (ou "Invite de commande" sous Windows) :

```
E:\> cd Code
E:\Code> mkdir AppTest
```

Cela crée un sous-dossier "AppTest" dans mon répertoire "E:\Code" qui sert pour tester différents trucs.

Ouvrir le dossier sous Visual Code

Toujours en ligne de commande, lancer Visual Code pour ouvrir le dossier "AppTest" :

```
E:\Code> cd AppTest
E:\Code\AppTest> code .
```

À partir de là, l'invite de commande de Windows ne sert plus à rien et peut être fermée. La suite se déroulera dans Visual Code ou dans son terminal.

Initialiser le projet Node

Pour cela, il faut ouvrir le terminal de Visual Code et lancer la commande `npm init` :

- Menu : View / Terminal
- Ou raccourci : Ctrl+ù

=>

```
PS E:\Code\AppTest> npm init -y
```

=>

```
Wrote to E:\Code\AppTest\package.json:

{
  "name": "AppTest",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

Note : Pour cet exemple, il est plus rapide de faire un `npm init -y` (ou `npm init --yes`) que de taper <Entrée> à chaque question pour accepter la valeur par défaut.

Dans Visual Code, il apparait maintenant le fichier "package.json" créé par NPM dans le dossier racine du projet ("E:\Code\AppTest" dans le cas présent).

2. Ajouter des modules au projet Node

Choix techniques

L'objectif de ce tutoriel est de tester le développement d'une application Node de type web. Pour cela, il faut donc installer Express car c'est le framework Node le plus utilisé pour ce genre d'application.

Express a besoin d'un système de templates pour générer les vues. Pour ne pas me compliquer la vie, je choisis EJS : il y a du vrai HTML dedans et ça ressemble énormément à la syntaxe d'ASP (d'avant Razor).

Pour gérer la base de données le plus simplement possible, SQlite sera suffisant. Surtout c'est ce qu'il y a de plus facile : pas de serveur à installer et zéro problème sous Windows. Avec Node JS, c'est le module SQlite3 qui sert d'interface pour SQlite.

Installer les dépendances

Cela se fait en ligne de commande, dans le terminal de Visual Code :

```
PS E:\Code\AppTest> npm install express
PS E:\Code\AppTest> npm install ejs
PS E:\Code\AppTest> npm install sqlite3
```

Ou pour aller plus vite :

```
PS E:\Code\AppTest> npm install express ejs sqlite3
```

Lorsque l'installation de ces trois dépendances (et de leurs propres dépendances) est terminée, le fichier "package.json" contient une nouvelle section "dependencies" qui enregistre la liste des dépendances du projet :

```
{
  "name": "AppTest",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
```

```
    "ejs": "^2.7.1",
    "express": "^4.17.1",
    "sqlite3": "^4.1.0"
  }
}
```

Note : Dans des tutoriels un peu anciens, on voit encore la syntaxe `npm install --save xxxxx` pour enregistrer la liste des dépendances dans le fichier "package.json", mais ce n'est plus nécessaire depuis la version 5 de NPM.

Le dossier "node_modules"

Le sous-répertoire "node_modules" est employé par NPM pour stocker tous les fichiers de dépendances d'un projet Node.

Lorsque le projet est versionné dans GIT, ce dossier doit être ignoré pour ne pas être commité dans le référentiel :

- C'est généralement un dossier énorme
- La commande `npm install` sans argument permet de (ré)installer les dépendances

Pour tester, on peut supprimer le dossier "node_modules" :

```
PS E:\Code\AppTest> rd node_modules /s /q
```

Note : Sous Windows, les options `/s /q` permettent de tout supprimer sans se poser de question.

Puis on installe toutes les dépendances listées dans le fichier "package.json" :

```
PS E:\Code\AppTest> npm install
```

3. Créer l'application Express

Vérifier que ça peut marcher...

Pour être certain que tout est installé correctement, le plus sûr est de commencer par un fichier "index.js" avec un contenu minimum :

```
const express = require("express");

const app = express();

app.listen(3000, () => {
  console.log("Serveur démarré (http://localhost:3000/) !");
});

app.get("/", (req, res) => {
  res.send("Bonjour le monde...");
});
```

Puis, dans le terminal de Visual Code :

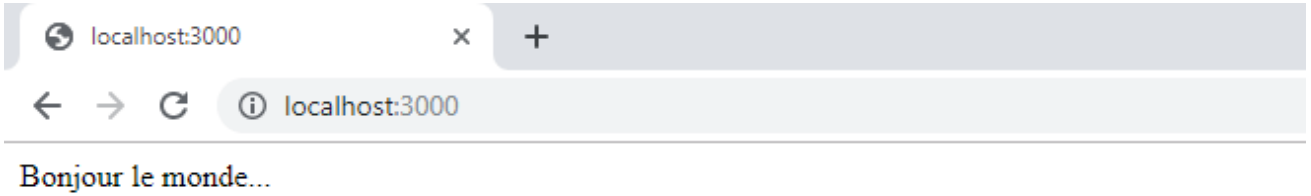
```
PS E:\Code\AppTest> node index
```

=>

```
Serveur démarré (http://localhost:3000/) !
```

Il ne reste plus qu'à contrôler que ça marche réellement :

- Lancer un navigateur
- Aller à l'URL "http://localhost:3000/"
- Le message "Bonjour le monde..." doit apparaitre comme ci-dessous :



C'est OK => arrêter le serveur en tapant Ctrl+C dans le terminal de Visual Code.

Comment ça marche ?

- La première ligne référence / importe le module Express.

```
const express = require("express");
```

- La ligne suivante sert à instancier un serveur Express.

```
const app = express();
```

- Ce serveur est ensuite démarré et attend les requêtes arrivant sur le port 3000. La fonction callback sert à afficher un message informatif lorsque le serveur est prêt à recevoir des requêtes.

```
app.listen(3000, () => {
  console.log("Serveur démarré (http://localhost:3000/) !");
});
```

- Ensuite vient une fonction pour répondre aux requêtes GET pointant sur la racine du site.

```
app.get("/", (req, res) => {
  res.send("Bonjour le monde...");
});
```

Grosso-modo...

Et plus précisément ?

Cela n'en a pas l'air, mais la méthode `app.get()` fait beaucoup de choses en seulement 3 lignes de codes.

Elle répond aux requêtes HTTP GET qui arrivent sur l'URL qui lui est passée en 1° paramètre. Dans notre cas, il s'agit de "/", c'est à dire la racine du site.

Lorsqu'une telle requête arrive, elle est passée à la fonction callback qui est définie en tant que 2° paramètre. Ici, il s'agit de la fonction arrow suivante :

```
(req, res) => {
  res.send("Bonjour le monde...");
}
```

Cette fonction callback reçoit en paramètres deux objets somme toute assez communs pour tout bon serveur web qui se respecte :

- la variable `req` qui contient un objet `Request`
- la variable `res` qui contient un objet `Response`

L'objet `Request` correspond à la requête HTTP qui a été envoyée par le navigateur (ou tout autre client). On peut donc y retrouver les informations relatives à cette requête, comme ses paramètres, les headers, les cookies, le body, etc...

L'objet `Response` correspond quant à lui à la réponse HTTP qui sera renvoyée au navigateur (ou à tout autre client) en bout de compte.

Dans notre programme, la réponse sera le texte "Bonjour le monde..." qui est envoyé grâce à la méthode `Response.send()`, qui fait "juste" deux trucs :

- Elle renvoie le texte dans la partie body de la réponse HTTP
- Elle met fin à la connection

Note : C'est quand même pas mal de technique pour ce tutoriel.

Améliorer le lancement de l'application Node JS

Revenons à des choses plus simples. Dans la section "scripts" du fichier "package.json", il est conseillé d'ajouter une ligne pour "automatiser" le lancement de l'application Node :

```
"start": "node index"
```

Ce qui donne (sans oublier la virgule en bout de ligne) :

```
{
  "name": "AppTest",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "start": "node index",
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
```

```
  "author": "",
  "license": "ISC",
  "dependencies": {
    "ejs": "^2.7.1",
    "express": "^4.17.1",
    "sqlite3": "^4.1.0"
  }
}
```

On peut maintenant lancer le programme par :

```
PS E:\Code\AppTest> npm start
```

=>

```
> AppTest@1.0.0 start E:\Code\AppTest
> node index.js

Serveur démarré (http://localhost:3000/) !
```

Et ne pas oublier le Ctrl+C pour arrêter le serveur Express à la fin.

Note : Il est possible d'utiliser le module "nodemon" pour ne plus devoir arrêter / redémarrer le serveur à chaque modification du code source. Mais je préfère éviter d'aborder trop de choses à la fois dans ce tutoriel.

4. Ajouter des vues EJS

Comme le but de l'application est d'avoir plusieurs fonctionnalités, on a besoin de créer plusieurs vues. Malheureusement, EJS ne gère pas les "layouts". Il faut donc louvoyer en insérant une vue partielle au début de la vue pour tout le HTML qui doit venir avant le contenu spécifique à la vue et

une seconde vue partielle avec le code HTML pour "terminer" la page.

Dans le cas de la vue correspondant à la requête vers la racine du site (soit un "GET /"), il faudra donc créer la vue "index.ejs" et les deux vues partielles réutilisables "_header.ejs" et "_footer.ejs".

Note : Ces trois fichiers doivent être enregistrés dans un dossier "views" qui doit donc être créé en premier lieu.

Vue partielle "views/_header.ejs"

```
<!doctype html>
<html lang="fr">

<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">
  <title>AppTest</title>
  <link rel="stylesheet" href="/css/bootstrap.min.css">
</head>

<body>

  <div class="container">

    <nav class="navbar navbar-expand-lg navbar-light bg-light">
      <a class="navbar-brand" href="/">AppTest</a>
      <ul class="navbar-nav mr-auto">
        <li class="nav-item">
          <a class="nav-link" href="/about">A propos</a>
        </li>
        <li class="nav-item">
          <a class="nav-link" href="/data">Données</a>
        </li>
        <li class="nav-item">
```

```
          <a class="nav-link" href="/livres">Livres</a>
        </li>
      </ul>
    </nav>
```

Vue "views/index.ejs"

```
<%- include("_header") -%>

<h1>Bonjour le monde...</h1>

<%- include("_footer") -%>
```

Vue partielle "views/_footer.ejs"

```
    <footer>
      <p>&copy; 2019 - AppTest</p>
    </footer>

  </div>

</body>

</html>
```

Note : À part les deux `<%- include(vue_partielle) -%>`, ce n'est que du HTML. C'est un des avantages de EJS par rapport à d'autres moteurs de template pour éviter d'avoir à se disperser quand on débute.

Ajouter une feuille de style

Comme on peut le voir dans les trois vues ci-dessus, elles font référence à Bootstrap 4.

Pour cela, il faut créer un dossier "public" dans lequel on crée ensuite un sous-dossier "css" où il suffit de copier le fichier "bootstrap.min.css" correspondant à la version 4.3.1 de Bootstrap dans mon cas.

5. Utiliser les vues dans Express

Note : Si cela n'avait pas été fait en début de projet, il aurait été nécessaire d'installer le module "EJS" par un `npm install ejs` pour pouvoir l'utiliser.

Modifications de "index.js"

Pour utiliser les vues créées dans l'application Express, il faut modifier quelque peu le fichier "index.js".

- Indiquer qu'il faut utiliser le moteur de template EJS.

```
app.set("view engine", "ejs");
```

Note : Il n'est pas nécessaire de faire un `const ejs = require("ejs")` avant car Express s'en charge pour nous.

- Indiquer que les vues sont enregistrées dans le dossier "views".

```
app.set("views", __dirname + "/views");
```

Ou mieux, en utilisant le module "path" inclus avec Node :

```
const path = require("path");
...
app.set("views", path.join(__dirname, "views"));
```

Note : Il n'y a pas besoin d'installer auparavant le module `path` par NPM, parce que c'est un module standard de Node JS.

- Indiquer que les fichiers statiques sont enregistrés dans le dossier "public" et ses sous-répertoires. C'est un paramétrage qui est nécessaire pour que le fichier "bootstrap.min.css" copié précédemment dans "public/css" soit accessible.

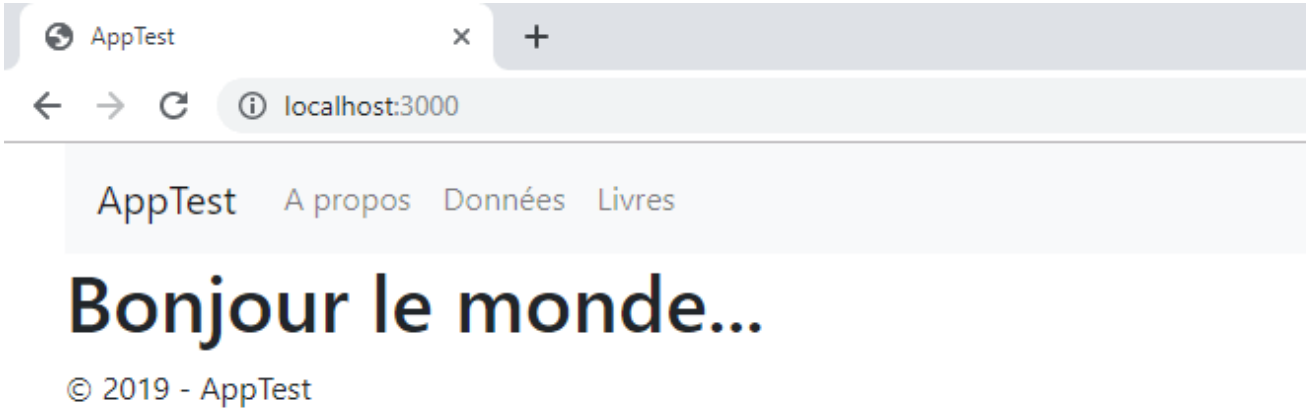
```
app.use(express.static(path.join(__dirname, "public")));
```

- Et enfin, renvoyer la vue "index.ejs" plutôt qu'un simple message "Bonjour le monde..." comme auparavant.

```
app.get("/", (req, res) => {
  // res.send("Bonjour le monde...");
  res.render("index");
});
```

Vérifier que cela fonctionne

- Faire un `npm start` dans le terminal de Visual Code
- Naviguer vers "http://localhost:3000/" avec Chrome
- La page suivante doit apparaitre :



Ajouter un chemin "/about"

La barre de navigation de l'application contient un choix "A propos" qui envoie vers l'URL "http://localhost:3000/about". Ce menu est défini dans la partie "nav" de la vue partielle "_header.ejs", mais pour l'instant, rien n'existe pour gérer cette route.

- Dans "index.js", ajouter une fonction pour répondre à une requête vers "/about" et renvoyer la vue "about.ejs" dans ce cas.

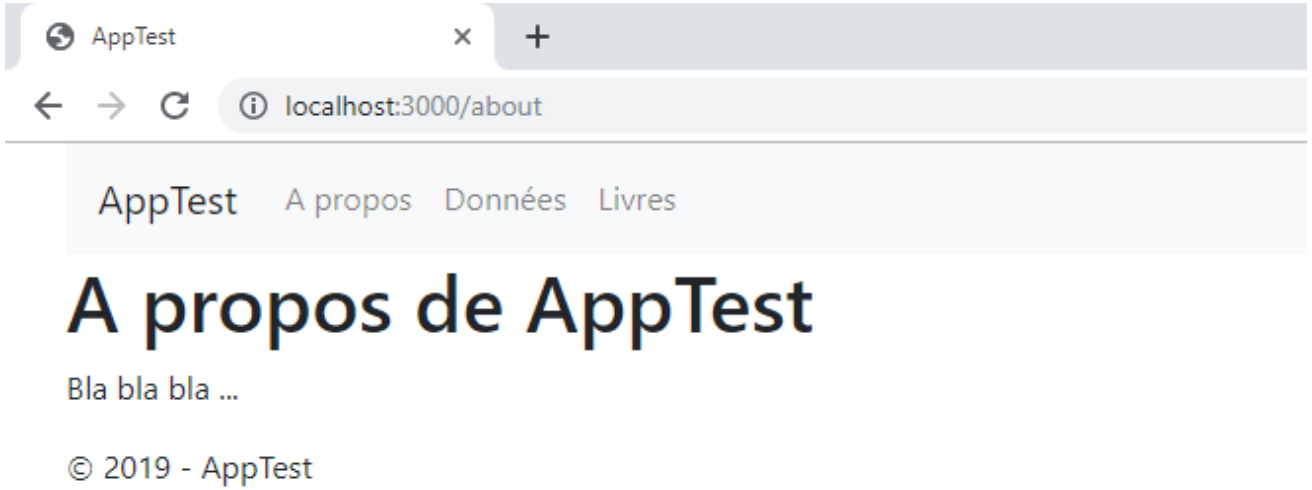
```
app.get("/about", (req, res) => {
  res.render("about");
});
```

- Créer une nouvelle vue "about.ejs" dans le dossier "views" (en ré-utilisant les deux vues partielles).

```
<%- include("_header") -%>

<h1>A propos de AppTest</h1>

<p>Bla bla bla ...</p>
```

```
<%- include("_footer") -%>
```

- Arrêter le serveur par Ctrl+C (si cela n'avait pas été fait plus tôt).
- Rédemarrer le serveur par `npm start` (c'est obligatoire pour prendre en compte les modifications apportées au projet).
- Naviguer vers "http://localhost:3000/".
- Cliquer sur le menu "A propos", ce qui donne :



Envoyer des données du serveur vers la vue

La barre de navigation de l'application contient aussi le choix "Données" qui envoie vers l'URL "http://localhost:3000/data". Cette URL va servir pour voir comment "injecter" des données dans la vue depuis le programme.

Tout d'abord, il faut ajouter une fonction à "index.js" pour prendre en compte l'URL "/data" et rendre la vue correspondante, mais cette fois-ci en indiquant en plus l'objet à lui transmettre.

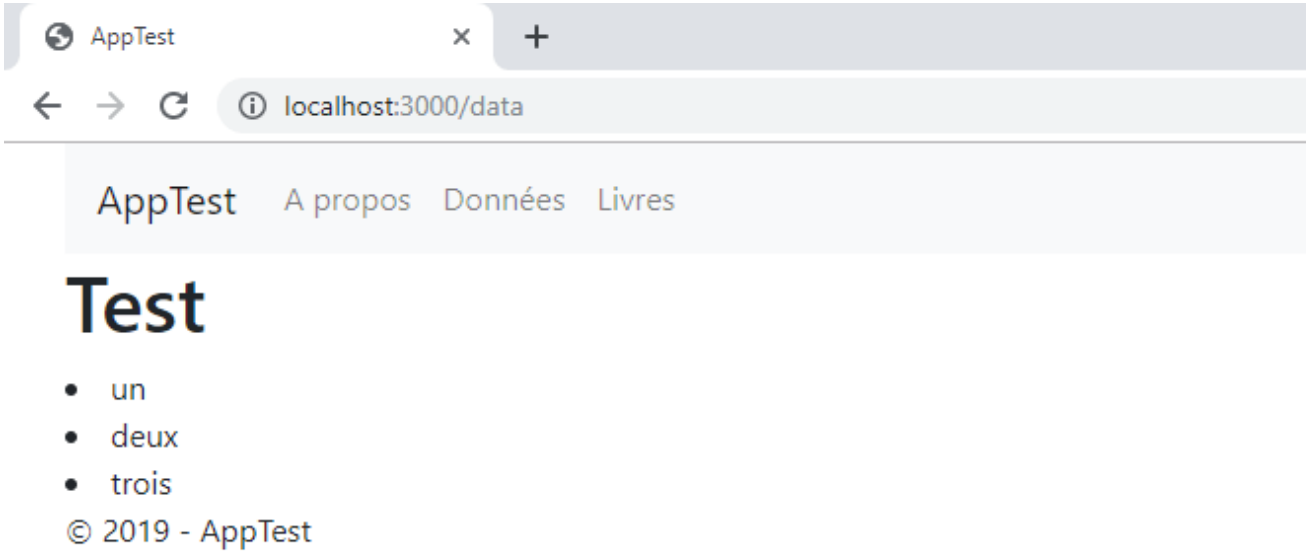
```
app.get("/data", (req, res) => {
  const test = {
    titre: "Test",
    items: ["un", "deux", "trois"]
  };
  res.render("data", { model: test });
});
```

Puis il faut ajouter une vue "data.ejs" dans le dossier "views" pour afficher les données qui lui sont transmises par l'application.

```
<%- include("_header") -%>

<h1><%= model.titre %></h1>

<ul>

  <% for (let i = 0; i < model.items.length; i++) { %>
    <li><%= model.items[i] %></li>
  <% } %>

</ul>

<%- include("_footer") -%>
```

Note : Le but de ce tutoriel n'est pas trop d'expliquer le fonctionnement d'EJS. J'ai justement choisi ce moteur de template parce que sa syntaxe à base de `<% ... %>` est assez répandue, que ce soit avec ASP, PHP, Ruby... Et pour le reste, c'est du JavaScript (d'où le nom Embedded JavaScript).

Et maintenant, quand on navigue vers "http://localhost:3000/data" après avoir redémarré le site, on obtient :



Le fichier "index.js" mis à jour

```
const express = require("express");
const path = require("path");

// Création du serveur Express
const app = express();

// Configuration du serveur
app.set("view engine", "ejs");
app.set("views", path.join(__dirname, "views"));
app.use(express.static(path.join(__dirname, "public")));

// Démarrage du serveur
app.listen(3000, () => {
    console.log("Serveur démarré (http://localhost:3000/) !");
});
```

```
// GET /
app.get("/", (req, res) => {
  // res.send("Bonjour le monde...");
  res.render("index");
});

// GET /about
app.get("/about", (req, res) => {
  res.render("about");
});

// GET /data
app.get("/data", (req, res) => {
  const test = {
    titre: "Test",
    items: ["un", "deux", "trois"]
  };
  res.render("data", { model: test });
});
```

6. Premiers pas avec le module SQlite3

Note : Si cela n'avait pas été fait en début de projet, il aurait été nécessaire d'installer le module SQlite3 par un `npm install sqlite3` pour pouvoir accéder à une base de données SQlite sous Node.

Déclarer le module SQlite3

Il faut commencer par référencer "sqlite3" en tête du programe "index.js", avec les deux autres déclarations pour "express" et "path".

```
const sqlite3 = require("sqlite3").verbose();
```

La méthode ".verbose()" permet d'avoir plus d'informations en cas de problème.

Connexion à la base de données SQlite

Ajouter ensuite le code pour se connecter à la base de données, juste avant de démarrer le serveur Express.

```
const db_name = path.join(__dirname, "data", "apptest.db");
const db = new sqlite3.Database(db_name, err => {
  if (err) {
    return console.error(err.message);
  }
  console.log("Connexion réussie à la base de données 'apptest.db'");
});
```

La base de données sera enregistrée dans le dossier "data", sous le nom "apptest.db". Elle est créée automatiquement si elle n'existe pas encore. Par contre, il est quand même nécessaire de créer le dossier "data" depuis Visual Code.

Après que ce code ait été exécuté, la variable "db" est un objet `Database` du module SQlite3 qui représente la connexion à la base de données. Cet objet va servir par la suite à accéder au contenu de la base de données et à effectuer des requêtes sur cette base de données.

Création d'une table "Livres"

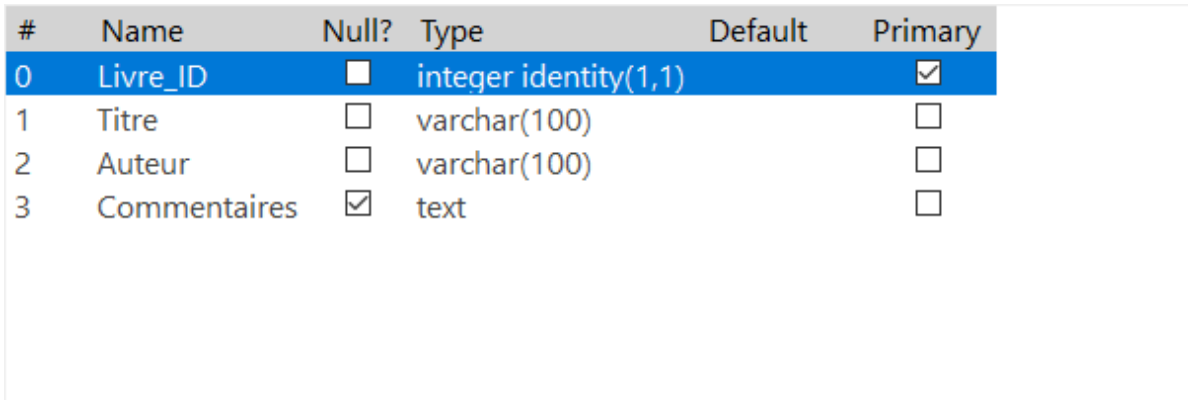
Pour ce tutoriel, on va créer une table de livres avec 4 colonnes :

- Livre_ID : l'identifiant automatique
- Titre : le titre du livre
- Auteur : l'auteur du livre
- Commentaires : un champ mémo avec quelques notes sur le livre

La requête SQL pour créer une telle table sous SQlite est la suivante :

```
CREATE TABLE IF NOT EXISTS Livres (
  Livre_ID INTEGER PRIMARY KEY AUTOINCREMENT,
  Titre VARCHAR(100) NOT NULL,
  Auteur VARCHAR(100) NOT NULL,
  Commentaires TEXT
);
```

Ce qui donne :



#	Name	Null?	Type	Default	Primary
0	Livre_ID	☐	integer identity(1,1)		☑
1	Titre	☐	varchar(100)		☐
2	Auteur	☐	varchar(100)		☐
3	Commentaires	☑	text		☐

Pour savoir comment on peut faire ça dans Node, on va créer la table depuis l'application. À cette fin, il suffit d'ajouter le code ci-dessous juste après s'être connecté à la base de données.

```
const sql_create = `CREATE TABLE IF NOT EXISTS Livres (
  Livre_ID INTEGER PRIMARY KEY AUTOINCREMENT,
  Titre VARCHAR(100) NOT NULL,
  Auteur VARCHAR(100) NOT NULL,
  Commentaires TEXT
);`;

db.run(sql_create, err => {
  if (err) {
```

```
    return console.error(err.message);
  }
  console.log("Création réussie de la table 'Livres'");
});
```

Ce code utilise la méthode `.run()` de l'objet `Database` du module SQlite3. Cette méthode exécute la requête SQL qui lui est passé en 1° paramètre puis appelle la fonction callback correspondant au 2° paramètre, en lui passant un objet `err` pour pouvoir vérifier si l'exécution de la requête s'est déroulée correctement.

Note : La table ne sera créée que si elle n'existe pas encore, grâce à la clause SQL "IF NOT EXISTS". Ça ne serait pas super pour une vraie application, mais là c'est juste un tutoriel.

Alimenter la table "Livres"

Pour faciliter la suite du tutoriel, il est plus pratique d'insérer dès maintenant quelques livres dans la base de données. Sous SQlite, on pourrait passer la requête suivante :

```
INSERT INTO Livres (Livre_ID, Titre, Auteur, Commentaires) VALUES
(1, 'Mrs. Bridge', 'Evan S. Connell', 'Premier de la série'),
(2, 'Mr. Bridge', 'Evan S. Connell', 'Second de la série'),
(3, 'L''ingénue libertine', 'Colette', 'Minne + Les égarements de Minne');
```

Si on a pas de client SQlite sous la main, c'est faisable en JavaScript, juste après avoir crée la table "Livres" (parce qu'il ne faut pas que l'insertion des livres ait lieu avant la création la table) :

```
  ...
  console.log("Création réussie de la table 'Livres'");
  // Alimentation de la table
  const sql_insert = `INSERT INTO Livres (Livre_ID, Titre, Auteur, Commentaires) VALUES
  (1, 'Mrs. Bridge', 'Evan S. Connell', 'Premier de la série'),
```

```
(2, 'Mr. Bridge', 'Evan S. Connell', 'Second de la série'),
(3, 'L''ingénue libertine', 'Colette', 'Minne + Les égarements de Minne');`;
db.run(sql_insert, err => {
  if (err) {
    return console.error(err.message);
  }
  console.log("Alimentation réussie de la table 'Livres'");
});
```

Normalement, il n'est pas utile de définir les identifiants lors des INSERT, mais dans ce cas, cela permet d'éviter que les données soient ré-insérées à chaque fois que le serveur démarre.

La première fois, la console affiche "Alimentation réussie de la table 'Livres'" et les fois suivantes l'erreur "SQLITE_CONSTRAINT: UNIQUE constraint failed: Livres.Livre_ID" puisque les 3 lignes existent déjà.

À présent, la table "Livres" contient les 3 lignes suivantes :

Livre_ID	Titre	Auteur	Commentaires
1	Mrs. Bridge	Evan S. Connell	Premier de la série
2	Mr. Bridge	Evan S. Connell	Second de la série
3	L'ingénue libertine	Colette	Minne + Les égarements de Minne

Afficher la liste des livres

Maintenant que la table "Livres" contient quelques données, il est possible de créer une méthode pour l'URL "http://localhost:3000/livres" du site de façon à lire la liste des livres enregistrés dans la base de données et à afficher cette liste dans la vue.

Pour lire la liste des livres, c'est assez simple. On fait une requête du style "SELECT * FROM ..." que l'on exécute via la méthode `db.all()` du module SQlite3. Une fois la requête terminée, cette méthode `db.all()` appelle une fonction callback en lui passant éventuellement une erreur et la liste des résultats obtenus par la requête SQL. Si tout va bien, la fonction callback peut alors transmettre ces résultats à la vue.

```
app.get("/livres", (req, res) => {
  const sql = "SELECT * FROM Livres ORDER BY Titre";
  db.all(sql, [], (err, rows) => {
    if (err) {
      return console.error(err.message);
    }
    res.render("livres", { model: rows });
  });
});
```

Quelques explications sur la ligne de code `db.all(sql, [], (err, rows) => { ... }` :

- Le 1° paramètre est la requête SQL à exécuter
- Le 2° paramètre est un tableau avec les variables nécessaires à la requête. Ici, la valeur "[]" est employée parce que la requête n'a pas besoin de variable.
- Le 3° paramètre est une fonction callback appelée après l'exécution de la requête SQL.
- "(err, rows)" correspond aux paramètres passés à la fonction callback. "err" contient éventuellement un objet erreur et "rows" est un tableau contenant la liste des lignes renvoyées par le SELECT.

Pour afficher cette liste de livres, on peut dans un premier temps créer une vue "livres.ejs" dans le dossier "views" avec le code suivant :

```
<%- include("_header") -%>

<h1>Liste des livres</h1>

<ul>
```

```
  <% for (const book of model) { %>
    <li>
      <%= book.Titre %>
      <em>(<%= book.Auteur %>)</em>
    </li>
  <% } %>

</ul>

<%- include("_footer") -%>
```

Après avoir relancé l'application par `npm start`, on obtient le résultat suivant en cliquant sur le menu "Livres" :

Note : Il faut faire attention et bien écrire "book.Titre" et pas "book.titre" parce que la table "Livres" a été créée en utilisant des majuscules comme initiales pour les noms des colonnes.

Afficher les livres sous forme de tableau

Maintenant que la méthode pour afficher la liste des livres fonctionne, on va améliorer la présentation de ces données. La vue de l'étape précédente utilisait une simple liste "ul / li" pour afficher les livres. Le code de cette vue "livres.ejs" va être totalement modifié pour employer une table HTML.

```
<%- include("_header") -%>

<h1>Liste des livres (<%= model.length %>)</h1>

<div class="table-responsive-sm">
  <table class="table table-hover">
    <thead>
      <tr>
        <th>Titre</th>
        <th>Auteur</th>
        <th>Commentaires</th>
        <th class="d-print-none">
          <a class="btn btn-sm btn-success" href="/create">Ajouter</a>
        </th>
      </tr>
    </thead>
    <tbody>
      <% for (const book of model) { %>
        <tr>
          <td><%= book.Titre %></td>
          <td><%= book.Auteur %></td>
          <td><%= book.Commentaires %></td>
          <td class="d-print-none">
            <a class="btn btn-sm btn-warning" href="/edit/<%= book.Livre_ID %>">Modifier</a>
            <a class="btn btn-sm btn-danger" href="/delete/<%= book.Livre_ID %>">Effacer</a>
          </td>
        </tr>
      <% } %>
    </tbody>
  </table>
</div>
```

```
<%- include("_footer") -%>
```

Et voilà ! Ctrl+C si nécessaire, `npm start` puis naviguer vers l'URL "http://localhost:3000/livres" pour avoir une vraie table Bootstrap.

L'avantage de cette nouvelle vue est de fournir des boutons [Ajouter], [Modifier] et [Effacer] pour mettre à jour la table des livres, ce qui est indispensable pour la suite du tutoriel.

7. Modifier une fiche

Cette partie du tutoriel va montrer comment modifier une fiche existante. On commencera par créer les vues nécéssaires pour saisir les informations du livre à modifier. Puis on codera une méthode servant à afficher le formulaire de saisie lorsque la route GET /edit/xxx sera appellée (via un clic sur le bouton [Modifier] dans la liste des livres). Et pour finir, une méthode correspondant à la route POST /edit/xxx servira à mettre à jour la base de données lorsque l'utilisateur validera les modifications apportées via le bouton [Modifier] en bas du formulaire de saisie.

Les vues "views/edit.ejs" et "views/_editor.ejs"

La vue principale pour pouvoir modifier une fiche est un formulaire Bootstrap assez classique.

```
<%- include("_header") -%>

<h1>Modifier une fiche</h1>

<form action="/edit/<%= model.Livre_ID %>" method="post">
  <div class="form-horizontal">

    <%- include("_editor") -%>

    <div class="form-group row">
      <label class="col-form-label col-sm-2"></label>
      <div class="col-sm-10">
        <input type="submit" value="Modifier" class="btn btn-default btn-warning" />
        <a class="btn btn-outline-dark cancel" href="/livres">Annuler</a>
      </div>
    </div>
  </div>
</form>

<%- include("_footer") -%>
```

La vue précédente fait appel à la vue partielle "_editor.ejs" qui contient le code HTML dédié aux différents champs de saisie. Cette vue partielle servira également un peu plus loin pour ajouter une nouvelle fiche.

```
<div class="form-group row">
  <label class="col-form-label col-sm-2" for="Titre">Titre</label>
  <div class="col-sm-8">
    <input autofocus class="form-control" name="Titre" value="<%= model.Titre %>" />
  </div>
</div>

<div class="form-group row">
  <label class="col-form-label col-sm-2" for="Auteur">Auteur</label>
  <div class="col-sm-7">
    <input class="form-control" name="Auteur" value="<%= model.Auteur %>" />
  </div>
</div>

<div class="form-group row">
  <label class="col-form-label col-sm-2" for="Commentaires">Commentaires</label>
  <div class="col-sm-10">
    <textarea class="form-control" cols="20" name="Commentaires" maxlength="32000" rows="7"><%= model.Commentaires %></textarea>
  </div>
</div>
```

La route GET /edit/xxx

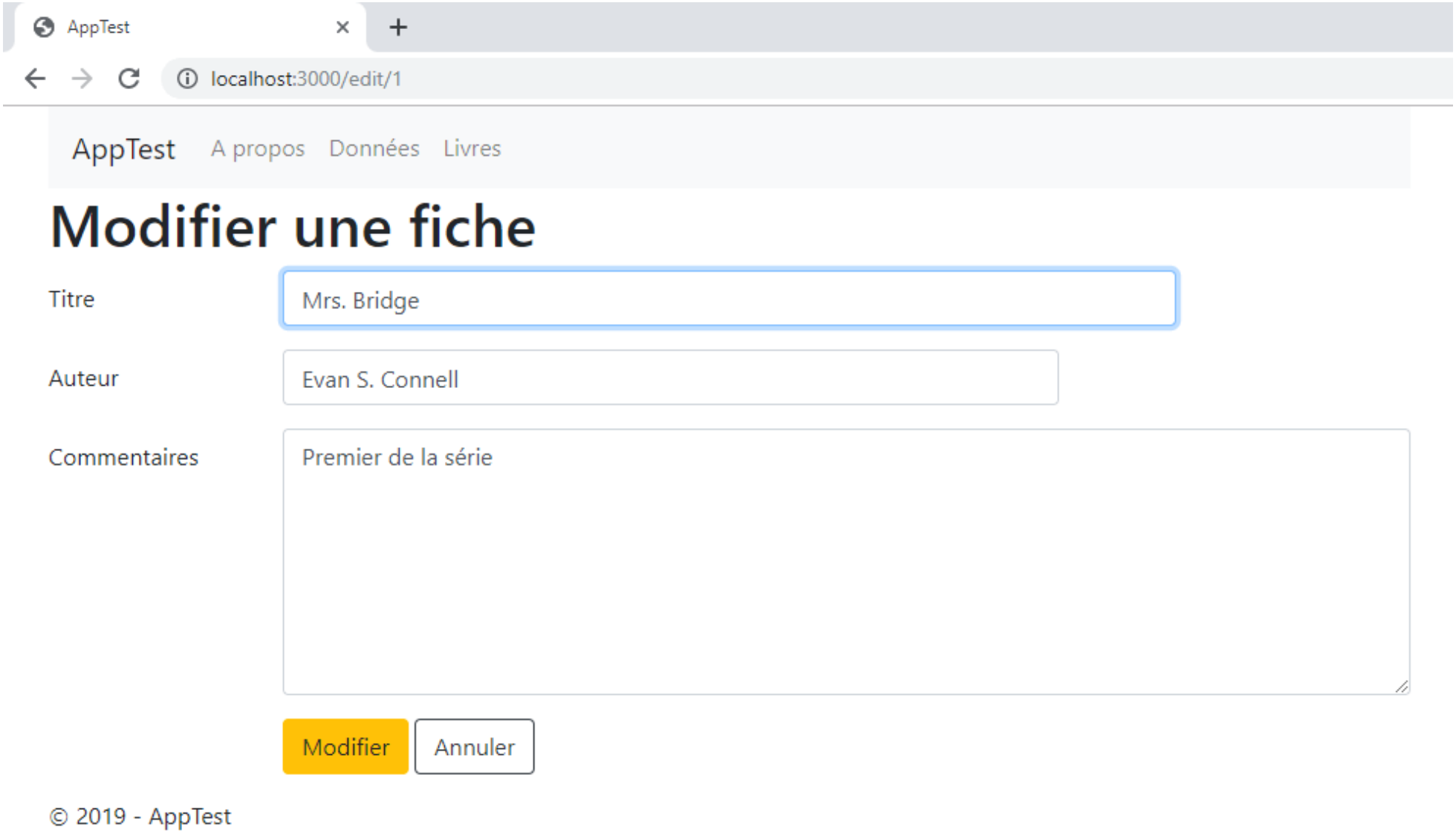
Il faut ensuite coder une première route pour afficher le livre à modifier lorsque on répond à la requête GET /edit/xxx (quand l'utilisateur a cliqué sur un bouton [Modifier] dans la liste des livres).

Pour cela, on défini l'URL à gérer sous la forme "/edit/:id" où ":id" correspond à l'identifiant de la fiche à modifier. Cet identifiant est récupérable via l'objet `Request` du framework Express, dans la liste de ses paramètres : `req.params.id`.

On peut alors faire une requête "SELECT ..." pour obtenir la fiche correspondant à cet identifiant. Cette requête est exécutée via la méthode `db.get()` de SQlite3 qui renvoie un seul résultat et qu'il est donc plus pratique d'utiliser que la méthode `db.all()` lorsqu'on fait un SELECT par identifiant. Dans ce cas, on passe en 2° paramètre l'identifiant du livre à afficher parce qu'on a utilisé une requête paramétrée (via le "... = ?") pour éviter l'injection SQL. Lorsque la requête est terminée, la fonction callback peut à son tour transmettre le résultat obtenu à la vue.

```
// GET /edit/5
app.get("/edit/:id", (req, res) => {
  const id = req.params.id;
  const sql = "SELECT * FROM Livres WHERE Livre_ID = ?";
  db.get(sql, id, (err, row) => {
    // if (err) ...
    res.render("edit", { model: row });
  });
});
```

Après redémarrage du serveur, voici le formulaire de saisie qui s'affiche désormais lorsque l'utilisateur clique sur un bouton [Modifier] dans la liste des livres :



La route POST /edit/xxx

Et pour finir, il ne reste plus qu'à coder la route pour sauvegarder les modifications apportées à la fiche, lors de la requête POST /edit/xxx. Le "post" se produit lorsque l'utilisateur valide sa saisie en cliquant sur le bouton [Modifier] du formulaire de saisie.

Là aussi, l'identifiant est retrouvé via le paramètre "id" de l'objet `Request`. Et les données saisies sont disponibles via la propriété `body` de cet objet `Request` pour être stockées dans un tableau temporaire avec l'identifiant.

Note : pour que `Request.body` récupère les valeurs postées, il est nécessaire d'ajouter un middleware à la configuration du serveur. Ce point sera expliqué plus précisément dans la partie suivante...

La modification en base de donnée se fait via une requête "UPDATE ..." exécutée avec la méthode `db.run()` de SQlite3 à laquelle on passe également le tableau contenant les donnés modifiées et l'identifiant du livre à mettre à jour.

Après avoir exécuté la requête "UPDATE ..." avec la méthode `db.run()` de SQlite3, la fonction callback redirige l'utilisateur vers la liste des livres à l'aide de la méthode `Response.redirect()` d'Express.

```
// POST /edit/5
app.post("/edit/:id", (req, res) => {
  const id = req.params.id;
  const book = [req.body.Titre, req.body.Auteur, req.body.Commentaires, id];
  const sql = "UPDATE Livres SET Titre = ?, Auteur = ?, Commentaires = ? WHERE (Livre_ID = ?)";
  db.run(sql, book, err => {
    // if (err) ...
    res.redirect("/livres");
  });
});
```

Note : Dans le cadre d'une vraie application, il faudrait impérativement avoir un contrôle de saisie côté client et côté serveur, mais ce n'est pas le sujet de ce tutoriel.

Le middleware "express.urlencoded()"

Comme évoqué dans la partie précédente, il est nécessaire d'utiliser le middleware "express.urlencoded()" pour que `Request.body` récupère les valeurs postées. Cela se fait tout simplement par un `app.use()` lors de la configuration du serveur.

```
// Configuration du serveur
app.set("view engine", "ejs");
app.set("views", path.join(__dirname, "views"));
app.use(express.static("public"));
app.use(express.urlencoded({ extended: false })); // <--- paramétrage du middleware
```

Ce middleware permet de récupérer les données envoyées en tant que "Content-Type: application/x-www-form-urlencoded", ce qui est le standard pour les valeurs postées depuis un formulaire. Pour information, on l'utilise très souvent avec le middleware "express.json()" pour les données envoyées en tant que "Content-Type: application/json", mais ici ce n'est pas nécessaire.

Note : Il y a des exemples qui utilisent encore le module "body-parser" à la place, mais ce n'est plus utile depuis la version 4.1.6 de Express.

8. Créer une nouvelle fiche

La vue "views/create.ejs"

La vue principale pour enregistrer un nouveau livre ressemble beaucoup à la vue codée pour modifier une fiche. Comme elle, elle fait appel à la vue partielle "_editor.ejs" pour ce qui concerne les différents champs de saisie.

```
<%- include("_header") -%>

<h1>Ajouter une fiche</h1>

<form action="/create" method="post">
  <div class="form-horizontal">

    <%- include("_editor") -%>
```

```
    <div class="form-group row">
      <label class="col-form-label col-sm-2"></label>
      <div class="col-sm-10">
        <input type="submit" value="Ajouter" class="btn btn-default btn-success" />
        <a class="btn btn-outline-dark cancel" href="/livres">Annuler</a>
      </div>
    </div>
  </div>
</form>

<%- include("_footer") -%>
```

La route GET /create

Par rapport à la modification, cette fonction est beaucoup plus simple. Elle se contente de renvoyer la vue "create.ejs" en lui transmettant un objet "livre" vide (parce que la vue partielle "_editor.ejs" attend un tel objet).

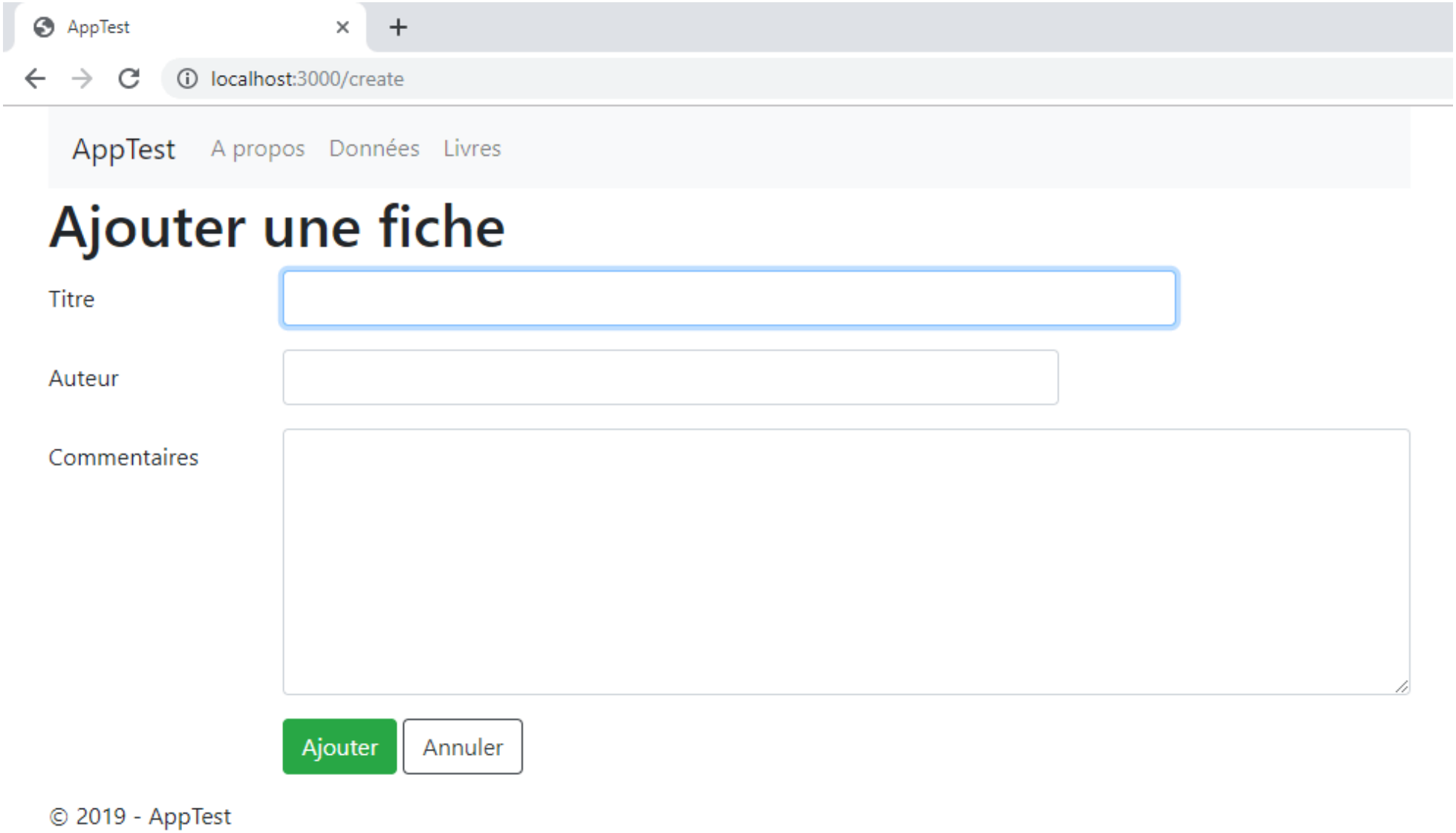
```
// GET /create
app.get("/create", (req, res) => {
  res.render("create", { model: {} });
});
```

Dans le cas d'une table plus riche que la table "Livres", il serait possible de définir des valeurs par défaut en codant cette méthode de la façon suivante :

```
// GET /create
app.get("/create", (req, res) => {
  const book = {
    Auteur: "Victor Hugo"
  }
```

```
  res.render("create", { model: book });
});
```

Comme on peut le voir ci-dessous, le formulaire de saisie pour ajouter un nouveau livre ressemble pas mal à celui pour modifier une fiche. C'est un des avantages de la vue partielle "_editor.ejs".



La route POST /create

Lorsque l'utilisateur clique sur le bouton [Ajouter] pour valider son formulaire de saisie, le navigateur envoie une requête "post" vers cette route. La méthode qui lui est associée ressemble beaucoup à celle mise en place pour la modification d'une fiche :

- Elle récupère les données saisies via la propriété `body` de l'objet `Request` du framework Express.
- La méthode `db.run()` de SQlite3 sert pour exécuter une requête "INSERT INTO ...".
- La fonction callback redirige l'utilisateur vers la liste des livres.

```
// POST /create
app.post("/create", (req, res) => {
  const sql = "INSERT INTO Livres (Titre, Auteur, Commentaires) VALUES (?, ?, ?)";
  const book = [req.body.Titre, req.body.Auteur, req.body.Commentaires];
  db.run(sql, book, err => {
    // if (err) ...
    res.redirect("/livres");
  });
});
```

9. Supprimer une fiche

Les vues "views/delete.ejs" et "views/_display.ejs"

La vue principale pour pouvoir supprimer une fiche doit en premier lieu afficher les informations du livre sélectionné pour permettre à l'utilisateur de confirmer sa suppression en toute connaissance. Elle ressemble par conséquent beaucoup aux vues "edit.ejs" et "create.ejs".

```
<%- include("_header") -%>

<h1>Effacer une fiche ?</h1>

<form action="/delete/<%= model.Livre_ID %>" method="post">
  <div class="form-horizontal">
```

```
    <%- include("_display") -%>

    <div class="form-group row">
      <label class="col-form-label col-sm-2"></label>
      <div class="col-sm-10">
        <input type="submit" value="Effacer" class="btn btn-default btn-danger" />
        <a class="btn btn-outline-dark cancel" href="/livres">Annuler</a>
      </div>
    </div>
  </div>
</form>

<%- include("_footer") -%>
```

Cette vue fait appel à la vue partielle "_display.ejs" qui contient le code HTML pour afficher les différents informations d'un livre. Pratiquement, ce code est quasi identique à celui de la vue "_editor.ejs", si ce n'est que les champs de saisie sont en "readonly".

```
<div class="form-group row">
  <label class="col-form-label col-sm-2" for="Titre">Titre</label>
  <div class="col-sm-8">
    <input readonly class="form-control" id="Titre" value="<%= model.Titre %>" />
  </div>
</div>

<div class="form-group row">
  <label class="col-form-label col-sm-2" for="Auteur">Auteur</label>
  <div class="col-sm-7">
    <input readonly class="form-control" id="Auteur" value="<%= model.Auteur %>" />
  </div>
</div>

<div class="form-group row">
```

```
  <label class="col-form-label col-sm-2" for="Commentaires">Commentaires</label>
  <div class="col-sm-10">
    <textarea readonly class="form-control" cols="20" id="Commentaires" maxlength="32000" rows="7"><%= model.Commentaires %></textarea>
  </div>
</div>
```

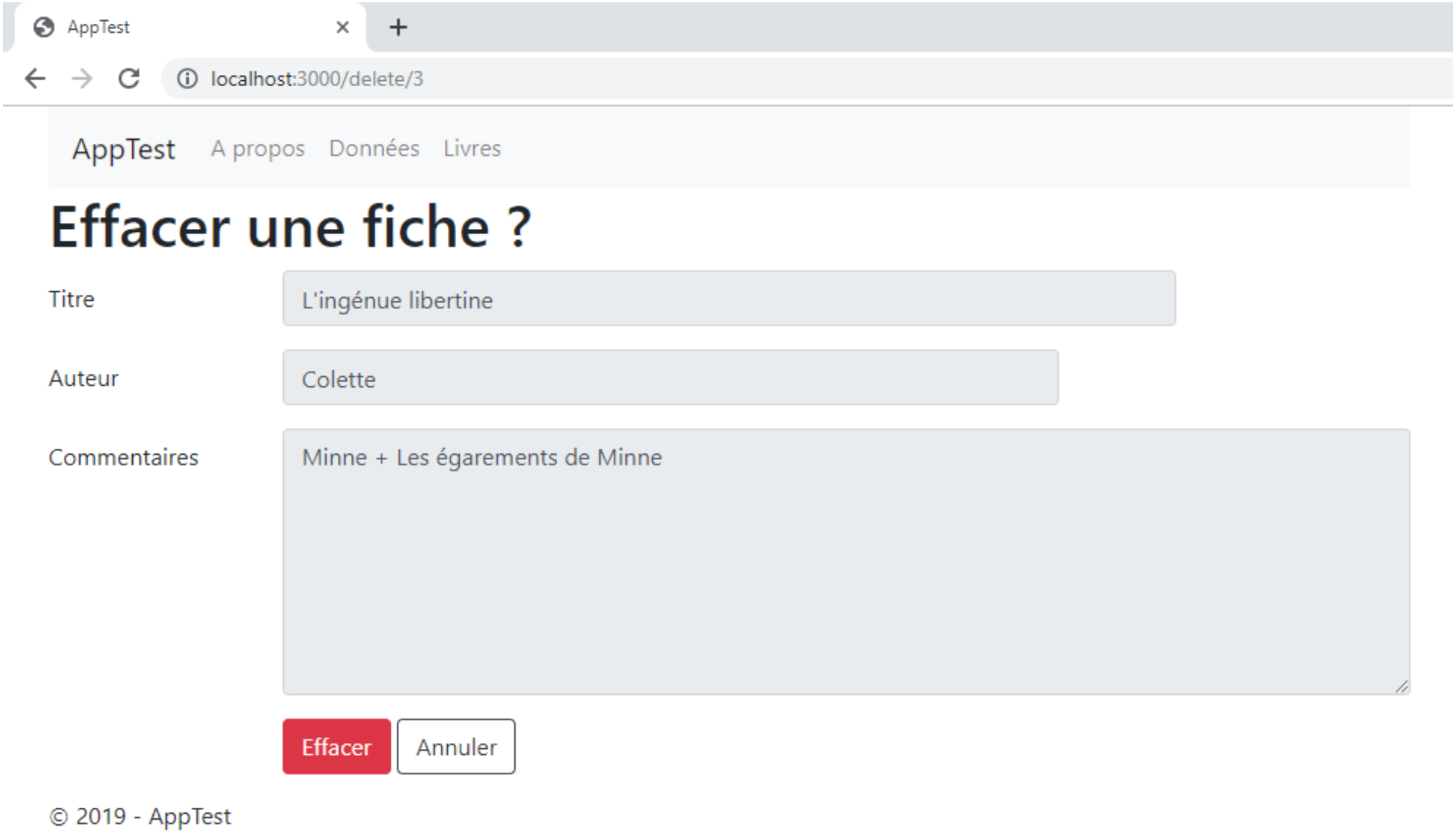
Si la table "Livres" contenait plus de colonnes que ce qu'il est possible d'afficher horizontalement dans la liste des livres, cette vue "_display.ejs" pourrait également être utilisée dans le cadre d'une route et d'une vue "details" qui serviraient à afficher l'intégralité de la fiche.

La route GET /delete/xxx

C'est le même code que la méthode GET /edit/xxx, si ce n'est qu'il renvoie la vue "delete.ejs" plutôt que la vue "edit.ejs".

```
// GET /delete/5
app.get("/delete/:id", (req, res) => {
  const id = req.params.id;
  const sql = "SELECT * FROM Livres WHERE Livre_ID = ?";
  db.get(sql, id, (err, row) => {
    // if (err) ...
    res.render("delete", { model: row });
  });
});
```

L'interface utilisateur est assez proche du formulaire de saisie habituel. Assez ironiquement, les trois zones de saisie ne sont en fait pas saisissables (et donc grisés selon les conventions de Bootstrap) :



La route POST /delete/xxx

Cette fonction toute simple répond à la requête "post" envoyée par le navigateur suite au clic sur le bouton [Effacer] pour confirmer la suppression du livre. Son code ressemble à pas mal de ce qui a déjà été vu jusqu'ici :

- Elle retrouve l'identifiant du livre à supprimer via `req.params.id`.
- La méthode `db.run()` de SQlite3 exécute une requête "DELETE ..." pour cet identifiant.
- La fonction callback redirige l'utilisateur vers la liste des livres.

```
// POST /delete/5
app.post("/delete/:id", (req, res) => {
  const id = req.params.id;
  const sql = "DELETE FROM Livres WHERE Livre_ID = ?";
  db.run(sql, id, err => {
    // if (err) ...
    res.redirect("/livres");
  });
});
```

10. Conclusion

Personnellement, ce tutoriel m'a permis de bien avancer. J'ai enfin écrit une application web permettant de mettre à jour une base de données SQL avec Node JS qui ressemble à ce que je peux faire avec Sinatra pour de petits trucs. Cela m'a donné un bon aperçu de tout ce qui est nécessaire et de voir que finalement ce n'est pas très éloigné de ce dont j'ai l'habitude avec ASP.NET MVC ou Sinatra.

Plus généralement, pour le côté Node, ce tutoriel a donné l'occasion de réviser un peu l'utilisation de NPM et son impact sur le fichier "package.json".

- npm init et npm init -y pour initialiser un projet
- npm install ... (sans --save) pour installer des modules
- npm start pour lancer le projet

Même si ce tutoriel n'a fait qu'effleurer ce que permet le framework Express, l'application développée constitue un bon point d'entrée pour s'entrainer avec une partie des méthodes offertes par Express. Au final, cela suffit pour réussir à organiser une application basique à la Sinatra.

- app.set(...) et app.use(...) pour configurer le serveur et les middlewares
- app.listen(port, callback) pour démarrer le serveur
- app.get(url, callback) pour répondre aux requêtes GET
- app.post(url, callback) pour les POST depuis les formulaires de saisie
- req.params.* pour récupérer les paramètres nommés de l'URL (la route)
- req.body.* pour accéder aux données postées par le formulaire de saisie

En ce qui concerne les vues, quelques-unes des fonctionnalités de base ont été passées en revue.

- res.send("texte") pour renvoyer un texte
- res.render(view_name, model) pour renvoyer une vue
- res.redirect(url) pour rediriger l'utilisateur
- utilisation de vues partielles pour se simplifier le travail
- et EJS ressemble beaucoup à ce qui se fait avec ASP ou aux vues ERB de Sinatra

Côté base de données, le programme a montré comment gérer une base SQlite et que c'est suffisament simple pour commencer (du moins quand on connait SQL). Mais cela semble assez spécifique au module SQlite3 et il reste à voir comment faire avec PostgreSQL, MySQL, Oracle ou Sql Server... L'idéal serait d'avoir quelque chose comme ADO.NET (ou ODBC à la rigueur) avant de passer à un véritable ORM.

- new sqlite3.Database() pour se connecter à la base de données (voire la créer)
- db.run(sql, [params], callback) pour exécuter des requêtes de mise à jour
- db.all(sql, [params], callback) pour une requête SELECT renvoyant plusieurs lignes
- db.get(sql, [params], callback) pour un SELECT par identifant

Pour ce qui est de JavaScript lui-même, cette application a surtout eu l'avantage de mettre en pratique quelques-unes des "nouveautés" du langage.

- employer des fonctions arrows pour les callbacks
- déclarer des constantes à chaque fois que c'est possible (c'est à dire toujours dans le programme développé)

- utiliser des boucles for ... of plus simples que des boucles classiques for (let i = 0; i < liste.length; i++)

Annexe - Le code complet de "index.js"

Ce n'est pas pour *rallonger* le billet, mais pour ceux qui comme moi aiment avoir une vue d'ensemble d'un programme. Et autant en profiter pour mettre en avant quelques petits chiffres :

- 148 lignes de codes
- 3 dépendances NPM (ejs, express et sqlite3)
- 3 modules importés (express, path et sqlite3)

Note : Le code complet de l'application est également disponible sur GitHub.

```
const express = require("express");
const path = require("path");
const sqlite3 = require("sqlite3").verbose();

// Création du serveur Express
const app = express();

// Configuration du serveur
app.set("view engine", "ejs");
app.set("views", path.join(__dirname, "views"));
app.use(express.static(path.join(__dirname, "public")));
app.use(express.urlencoded({ extended: false }));

// Connexion à la base de donnée SQlite
const db_name = path.join(__dirname, "data", "apptest.db");
const db = new sqlite3.Database(db_name, err => {
  if (err) {
    return console.error(err.message);
  }
```

```
  console.log("Connexion réussie à la base de données 'apptest.db'");
});

// Création de la table Livres (Livre_ID, Titre, Auteur, Commentaires)
const sql_create = `CREATE TABLE IF NOT EXISTS Livres (
  Livre_ID INTEGER PRIMARY KEY AUTOINCREMENT,
  Titre VARCHAR(100) NOT NULL,
  Auteur VARCHAR(100) NOT NULL,
  Commentaires TEXT
);`;
db.run(sql_create, err => {
  if (err) {
    return console.error(err.message);
  }
  console.log("Création réussie de la table 'Livres'");
  // Alimentation de la table
  const sql_insert = `INSERT INTO Livres (Livre_ID, Titre, Auteur, Commentaires) VALUES
  (1, 'Mrs. Bridge', 'Evan S. Connell', 'Premier de la série'),
  (2, 'Mr. Bridge', 'Evan S. Connell', 'Second de la série'),
  (3, 'L''ingénue libertine', 'Colette', 'Minne + Les égarements de Minne');`;
  db.run(sql_insert, err => {
    if (err) {
      return console.error(err.message);
    }
    console.log("Alimentation réussie de la table 'Livres'");
  });
});

// Démarrage du serveur
app.listen(3000, () => {
  console.log("Serveur démarré (http://localhost:3000/) !");
});

// GET /
app.get("/", (req, res) => {
```

```
  // res.send("Bonjour le monde...");
  res.render("index");
});

// GET /about
app.get("/about", (req, res) => {
  res.render("about");
});

// GET /data
app.get("/data", (req, res) => {
  const test = {
    titre: "Test",
    items: ["un", "deux", "trois"]
  };
  res.render("data", { model: test });
});

// GET /livres
app.get("/livres", (req, res) => {
  const sql = "SELECT * FROM Livres ORDER BY Titre";
  db.all(sql, [], (err, rows) => {
    if (err) {
      return console.error(err.message);
    }
    res.render("livres", { model: rows });
  });
});

// GET /create
app.get("/create", (req, res) => {
  res.render("create", { model: {} });
});

// POST /create
```

```
app.post("/create", (req, res) => {
  const sql = "INSERT INTO Livres (Titre, Auteur, Commentaires) VALUES (?, ?, ?)";
  const book = [req.body.Titre, req.body.Auteur, req.body.Commentaires];
  db.run(sql, book, err => {
    if (err) {
      return console.error(err.message);
    }
    res.redirect("/livres");
  });
});

// GET /edit/5
app.get("/edit/:id", (req, res) => {
  const id = req.params.id;
  const sql = "SELECT * FROM Livres WHERE Livre_ID = ?";
  db.get(sql, id, (err, row) => {
    if (err) {
      return console.error(err.message);
    }
    res.render("edit", { model: row });
  });
});

// POST /edit/5
app.post("/edit/:id", (req, res) => {
  const id = req.params.id;
  const book = [req.body.Titre, req.body.Auteur, req.body.Commentaires, id];
  const sql = "UPDATE Livres SET Titre = ?, Auteur = ?, Commentaires = ? WHERE (Livre_ID = ?)";
  db.run(sql, book, err => {
    if (err) {
      return console.error(err.message);
    }
    res.redirect("/livres");
  });
});
```

```
// GET /delete/5
app.get("/delete/:id", (req, res) => {
  const id = req.params.id;
  const sql = "SELECT * FROM Livres WHERE Livre_ID = ?";
  db.get(sql, id, (err, row) => {
    if (err) {
      return console.error(err.message);
    }
    res.render("delete", { model: row });
  });
});

// POST /delete/5
app.post("/delete/:id", (req, res) => {
  const id = req.params.id;
  const sql = "DELETE FROM Livres WHERE Livre_ID = ?";
  db.run(sql, id, err => {
    if (err) {
      return console.error(err.message);
    }
    res.redirect("/livres");
  });
});
```

English version: CRUD application with Express and SQlite in 10 steps.