**Utilisation de Sequelize.js et SQLite dans une application REST API Express.js**

Dans ce didacticiel, nous allons créer une application Web simple de gestion des contacts à l'aide de Node.js , Express.js , Vue.js en conjonction Sequelize.js et une [base de](https://translate.googleusercontent.com/translate_c?depth=1&hl=en&prev=_dd&rurl=translate.google.com&sl=en&sp=nmt4&tl=fr&u=https://www.sqlite.org/index.html&xid=17259,15700021,15700124,15700149,15700186,15700190,15700201&usg=ALkJrhhGfWGmcluuPCQA8DsF_EQAWqbrqg) données SQLite .

### **Configuration et installation**

1. Créer le répertoire node-vue-sqlite-sequelize
2. Ouvrir le dossier crée et taper dans le terminal : npm init pour initialiser l’application
3. Installer les modules dont nous avons besoin :

npm install --save express body-parser sequelize sequelize-cli sqlite3 nodemon

### **Création de l'interface utilisateur de gestion des contacts**

1. Créer le dossier static dans le dossier de l’application
2. A l’intérieur de ce nouveau dossier, créer le fichier index.html

Étant donné que l'application gère les contacts, nous aurons besoin d'une représentation d'une personne, qui va être l'objet Contact. En ce qui concerne les comportements, nous aurons besoin des éléments suivants:

* Voir la liste de tous les contacts
* Ajouter des contacts
* Afficher les détails d'un contact
* Mettre à jour les détails d'un contact
* Supprimer un contact

Nous interagissons avec le serveur via une API AJAX REST. Par conséquent, nous aurons besoins des méthodes suivantes :

| Route | Méthode | La fonctionnalité |
| --- | --- | --- |
| /api/contacts | GET | Retrouve tous les contacts |
| /api/contacts | POST | Créer un contact |
| /api/contacts/:id | PUT | Mettre à jour les détails d'un contact |
| /api/contacts/:id | DELETE | Supprimer un contact |

Le contenu du fichier index.html est le suivant :

<!-- index.html -->

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<meta http-equiv="X-UA-Compatible" content="ie=edge">

<title>Contacts</title>

<link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/bulma/0.7.1/css/bulma.css">

<script src="https://cdnjs.cloudflare.com/ajax/libs/vue/2.5.17-beta.0/vue.js"></script>

<script src="https://unpkg.com/axios/dist/axios.min.js"></script>

<link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/4.7.0/css/font-awesome.min.css">

<style>

.section-container {

max-width: 800px;

margin-right: auto;

margin-left: auto;

}

</style>

</head>

<body>

<div id="app" class="container">

<section class="section section-container" style="padding-top: 24px; padding-bottom: 5px;">

<h2 class="title">Contacts</h2>

<contact v-for="contact in contacts"

:key="contact.name"

:contact="contact"

@save-contact="onAddOrUpdateContact"

@delete-contact="deleteContact" />

</section>

<section class="section section-container" style="padding-bottom: 10px;">

<div class="box">

<add-update-contact title="Add Contact" @save-contact="onAddOrUpdateContact" />

</div>

</section>

</div>

<script>

const AddUpdateContact = {

props: ['contact', 'title'],

data () {

return {

id: this.contact ? this.contact.id : null,

firstName: this.contact ? this.contact.firstName : '',

lastName: this.contact ? this.contact.lastName : '',

phone: this.contact ? this.contact.phone : ''

}

},

methods: {

save() {

this.$emit('save-contact', { id: this.id, firstName: this.firstName, lastName: this.lastName, phone: this.phone })

if (!this.id) {

this.firstName = ''

this.lastName = ''

this.phone = ''

}

}

},

template: `

<form class="form" @submit.prevent="save">

<h3 class='subtitle'>{{ title }}</h3>

<div class="field">

<label>First Name</label>

<div class="control">

<input class="input" type="text" v-model="firstName">

</div>

</div>

<div class="field">

<label>Last Name</label>

<div class="control">

<input class="input" type="text" v-model="lastName">

</div>

</div>

<div class="field">

<label>Phone</label>

<div class="control">

<input class="input" type="text" v-model="phone">

</div>

</div>

<div class="field">

<div class="control">

<button class="button is-success">Save</button>

</div>

</div>

</form>

`

}

const Contact = {

props: ['contact'],

components: { 'add-update-contact': AddUpdateContact },

data () {

return {

showDetail: false

}

},

methods: {

onAddOrUpdateContact(contact) {

this.$emit('save-contact', contact)

},

deleteContact (contact) {

this.$emit('delete-contact', contact)

}

},

template: `

<div class="card">

<header class="card-header">

<p @click="showDetail = !showDetail" class="card-header-title">

{{ contact.firstName }} {{ contact.lastName }}

</p>

<a class="card-header-icon" @click.stop="deleteContact(contact)">

<span class="icon">

<i class="fa fa-trash"></i>

</span>

</a>

</header>

<div v-show="showDetail" class="card-content">

<add-update-contact title="Details" :contact="contact" @save-contact="onAddOrUpdateContact" />

</div>

</div>

`

}

new Vue({

el: '#app',

components: { contact: Contact, 'add-update-contact': AddUpdateContact },

data: {

contacts: [],

apiURL: 'http://localhost:3000/api/contacts'

},

methods: {

onAddOrUpdateContact (contact) {

if (contact.id) {

this.updateContact(contact)

} else {

this.addContact(contact)

}

},

addContact (contact) {

return axios.post(this.apiURL, contact)

.then((response) => {

const copy = this.contacts.slice()

copy.push(response.data)

this.contacts = copy

})

},

updateContact (contact) {

return axios.put(`${this.apiURL}/${contact.id}`, contact)

.then((response) => {

const copy = this.contacts.slice()

const idx = copy.findIndex((c) => c.id === response.data.id)

copy[idx] = response.data

this.contacts = copy

})

},

deleteContact (contact) {

console.log('deleting', contact)

return axios.delete(`${this.apiURL}/${contact.id}`)

.then((response) => {

let copy = this.contacts.slice()

const idx = copy.findIndex((c) => c.id === response.data.id)

copy.splice(idx, 1)

this.contacts = copy

})

}

},

beforeMount () {

axios.get(this.apiURL)

.then((response) => {

this.contacts = response.data

})

}

})

</script>

</body>

</html>

### **Construction de l'API REST**

Nous utilisons Express.js pour cette application, nous allons crée un fichier nommé server.js dans le même répertoire que le fichier package.json, puis l'ouvrir dans l’éditeur.

En haut du fichier, on insère le framework Express.js via require , puis on instancie une instance d'application. Ensuite, on informe l’application d'utiliser le middleware statique de l'Express pour diffuser du contenu statique (HTML, JS, CSS, et ...) à partir du répertoire "static". Tout en bas du script server.js, on dit au serveur d'applications d’écouter le port 3000.

On utilisera body-parser pour un analyser les requêtes.

// server.js

const express = require('express');

const bodyParser = require('body-parser');

const app = express();

app.use(bodyParser.json());

app.use(express.static(\_\_dirname + '/static'));

app.get('/api/contacts', (req, res) => {

// TODO: retreive contacts and send to requester

});

app.post('/api/contacts', (req, res) => {

const { firstName, lastName, phone } = req.body

// TODO: create contact

});

app.delete('/api/contacts/:id', (req, res) => {

const id = parseInt(req.params.id)

// TODO: find and delete contact by id

});

app.put('/api/contacts/:id', (req, res) => {

const id = parseInt(req.params.id)

const { firstName, lastName, phone } = req.body

// TODO: find and update contact by id

});

app.listen(3000, () => {

console.log('Server is up on port 3000');

});

### **L’ORM Sequelize et le Sequelize CLI**

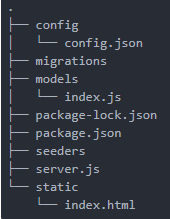
Sequelize.js est un ORM (Object Relational Mapper) populaire pour Node.js version 4 et ultérieure, qui peut être utilisé pour de nombreux systèmes de gestion de base de données (SGBD) tels que MySQL, Postgres, SQLite, etc. Il existe une bibliothèque d’utilitaires complémentaire appelée [sequelize-cli,](https://translate.googleusercontent.com/translate_c?depth=1&hl=en&prev=_dd&rurl=translate.google.com&sl=en&sp=nmt4&tl=fr&u=https://github.com/sequelize/cli&xid=17259,15700021,15700124,15700149,15700186,15700190,15700201&usg=ALkJrhgixNY-oGmi0wAzwgkiVK-oqG3UaA) qui permet d’automatiser certaines parties non triviales de la programmation de bases de données.

Dans ce didacticiel, on utilisera sequelize-cli pour générer le langage DDL (Data Definition Language) afin de créer des tables de base de données, ainsi que pour générer des modèles de données créant et exécutant le langage DML (Data Manipulation Language) permettant d'interroger la base de données. un système de migration permettant de contrôler le schéma de la base de données.

Pour commencer, on va utiliser sequelize-cli pour initialiser la couche d’accès aux données du projet de la manière suivante:

node\_modules/.bin/sequelize init

Cette commande créera les répertoires config, migrations, modèles et générateurs, ce qui donnera à la structure de mon projet l'aspect suivant:



Les répertoires ont pour but:

* config / index.js - définit les paramètres de connexion
* migrations - contient des scripts de migration permettant de gérer la gestion des versions du schéma
* models - contient les modèles de données que vous utilisez pour interagir avec la base de données dans le code de votre application
* seeders - contient des scripts pour remplir votre base de données avec les données initiales

Tout d'abord, on doit éditer le fichier config / config.json pour indiquer à Sequelize qu’on va utiliser une base de données SQLite. On modifiera celui-ci :

{

"development": {

"username": "root",

"password": null,

"database": "database\_development",

"host": "127.0.0.1",

"dialect": "mysql"

},

"test": {

"username": "root",

"password": null,

"database": "database\_test",

"host": "127.0.0.1",

"dialect": "mysql"

},

"production": {

"username": "root",

"password": null,

"database": "database\_production",

"host": "127.0.0.1",

"dialect": "mysql"

}

}

Par celui-là :

{

"development": {

"dialect": "sqlite",

"storage": "./database.sqlite3"

},

"test": {

"dialect": "sqlite",

"storage": ":memory"

},

"production": {

"dialect": "sqlite",

"storage": "./database.sqlite3"

}

}

Cela créera et utilisera un fichier de base de données SQLite appelé database.sqlite3 à la racine du projet.

On va suivre cette commande avec une autre, mais cette fois, on utilisera l’argument model:generate pour définir le modèle Contact et ses attributs, comme suit:

node\_modules/.bin/sequelize model:generate --name Contact --attributes firstName:string,lastName:string,phone:string,email:string

Le paramètre --name est évidemment le nom du modèle à générer et le paramètre --attibutes est suivi des champs d’objet qui le définissent, ainsi que de leurs types de données. Les sorties de cette commande sont deux nouveaux fichiers:

1. models / contact.js - un modèle de données à utiliser dans le code de logique d'application Node.js
2. migrations / yyyymmddHHMMSS-create-contact.js - un script de migration qui émettra DDL SQL pour créer la table des contacts dans la base de données

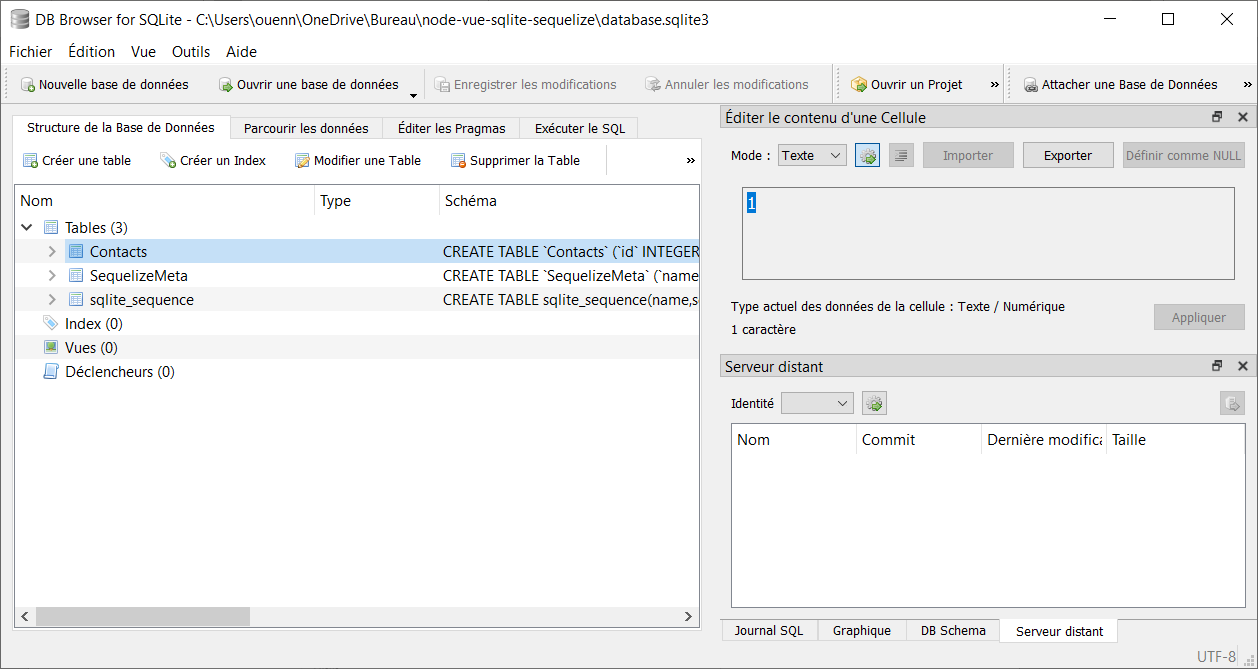
En plus des attributs spécifiés dans le model:generate commande generate, sequelize-cli générera également un champ d' id entier auto-incrémenté ainsi que des champs createdAt et updatedAt date-link.

La prochaine étape consiste à exécuter la migration de manière à ce que la base de données SQLite contienne la table de contacts comme suit:

node\_modules/.bin/sequelize db:migrate

Cette commande indiquera que la migration a été exécutée avec succès.

On peut maintenant ouvrir mon fichier database.sqlite3 nouvellement généré et afficher le schéma en utilisant l’utilitaire DB Browser for SQLite (https://download.sqlitebrowser.org/SQLiteDatabaseBrowserPortable\_3.11.2\_English.paf.exe):



Notez qu'il existe également une autre table nommée SequelizeMeta. Il s'agit de la table utilisée par Sequelize.js pour conserver l'ordre dans lequel les migrations sont exécutées.

Maintenant que le modèle Contact est mappé sur une table de base de données, on souhaite générer un script seeder pour peupler ma base de données avec des données expliquant comment interagir avec l'ORM dans mon code d'application. La génération d'un script seeder se fait par la commande suivante :

node\_modules/.bin/sequelize seed:generate --name seed-contact

La sortie est un nouveau script dans le répertoire de départ (seeders) avec la dénomination yyyymmddHHMMSS-seed-contact.js qu’on va modifier de la façon suivante :

'use strict';

module.exports = {

up: (queryInterface, Sequelize) => {

/\*

Add altering commands here.

Return a promise to correctly handle asynchronicity.

Example:

return queryInterface.bulkInsert('Person', [{

name: 'John Doe',

isBetaMember: false

}], {});

\*/

return queryInterface.bulkInsert('Contacts', [{

firstName: 'Snoop',

lastName: 'Dog',

phone: '111-222-3333',

email: 'snoopydog@dogpound.com',

createdAt: new Date().toDateString(),

updatedAt: new Date().toDateString()

}, {

firstName: 'Scooby',

lastName: 'Doo',

phone: '444-555-6666',

email: 'scooby.doo@misterymachine.com',

createdAt: new Date().toDateString(),

updatedAt: new Date().toDateString()

}, {

firstName: 'Herbie',

lastName: 'Husker',

phone: '402-437-0001',

email: 'herbie.husker@unl.edu',

createdAt: new Date().toDateString(),

updatedAt: new Date().toDateString()

}], {});

},

down: (queryInterface, Sequelize) => {

/\*

Add reverting commands here.

Return a promise to correctly handle asynchronicity.

Example:

return queryInterface.bulkDelete('Person', null, {});

\*/

return queryInterface.bulkDelete('Contacts', null, {});

}

};

Enfin, on doit exécuter le seeder pour remplir la base de données avec ces contacts initiaux.

node\_modules/.bin/sequelize db:seed:all

Ce qui donne une sortie dans la console pour me faire savoir que la table de base de données a été crée avec succès avec data.

Cependant, avant de pouvoir utiliser le modèle Contact, on doit informer l'application Express.js de son existence. On le fait en ajoutant un require dans server.js et en l’attribuant à une variable const appelée db comme ceci:

// server.js

const express = require('express');

const bodyParser = require('body-parser');

const db = require('./models'); // new require for db object

Cette variable db contiendra le modèle Contact, qui est accessible via db.Contact .

On commence par le plus simple des points de terminaison de l'API, le point de terminaison GET /api/contacts :

Ce module doit simplement renvoyer tous les contacts de la base de données et renvoyer une réponse au demandeur. Je peux appeler la méthode findAll de l'objet Contact et envoyer les contacts au client à l'aide de la commande res.send (...) bien connue fournie par le framework Express.js.

// server.js

// ommitting everything above ...

app.get('/api/contacts', (req, res) => {

return db.Contact.findAll()

.then((contacts) => res.send(contacts))

.catch((err) => {

console.log('There was an error querying contacts', JSON.stringify(err))

return res.send(err)

});

});

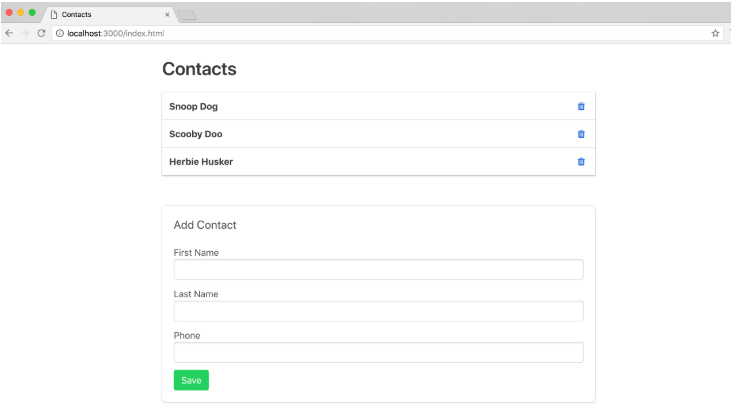
// omitting everything below...

On peut démarrer le serveur Express et pointer le navigateur sur localhost: 3000/index.html

npm start

ou

node server.js



Pour le point de terminaison POST /api/contacts :

Pour implémenter la fonctionnalité de création pour le point de terminaison POST / api / contacts, la création de nouvelles instances de contact est très simple. Appelez simplement la méthode create (...) de l'objet db.Contact, en enchaînant sur then (...) et je renverrai ensuite le contact nouvellement créé au client, comme suit:

// server.js

// ommitting everything above ...

app.post('/api/contacts', (req, res) => {

const { firstName, lastName, phone } = req.body

return db.Contact.create({ firstName, lastName, phone })

.then((contact) => res.send(contact))

.catch((err) => {

console.log('\*\*\*There was an error creating a contact', JSON.stringify(contact))

return res.status(400).send(err)

})

});

// omitting everything below ...

Pour le point de terminaison PUT /api/contacts/:id

Pour le point de terminaison PUT / api / contacts /: id. Il s'agit d'une interaction en deux parties du point de vue de la programmation de la base de données.

Tout d'abord, je commence par rechercher le contact correspondant à l'ID indiqué dans l'URL de l'API à l'aide de la méthode db.Contact.findByPk (...), puis je modifie les champs avec les nouvelles données et je termine par l'utilisation de la méthode update (...) de l'objet contact envoyant le modèle mis à jour à la base de données où ces modifications seront persistées.

// server.js

// ommitting everything above ...

app.put('/api/contacts/:id', (req, res) => {

const id = parseInt(req.params.id)

return db.Contact.findByPk(id)

.then((contact) => {

const { firstName, lastName, phone } = req.body

return contact.update({ firstName, lastName, phone })

.then(() => res.send(contact))

.catch((err) => {

console.log('\*\*\*Error updating contact', JSON.stringify(err))

res.status(400).send(err)

})

})

});

// omitting everything below...

Pour le point de terminaison DELETE /api/contacts/:id

Pour supprimer un contact, on utilise le point de terminaison DELETE / api / contacts /: id. La suppression est très similaire à la fonctionnalité de mise à jour en ce sens que l'instance de modèle que vous souhaitez supprimer doit tout d'abord être extraite de la base de données. Vous devez ensuite appeler la méthode destroy () de l'instance de modèle.

// server.js

// ommitting everything above ...

app.delete('/api/contacts/:id', (req, res) => {

const id = parseInt(req.params.id)

return db.Contact.findByPk(id)

.then((contact) => contact.destroy())

.then(() => res.send({ id }))

.catch((err) => {

console.log('\*\*\*Error deleting contact', JSON.stringify(err))

res.status(400).send(err)

})

});

// omitting everything below ...

L'intégralité du script server.js est présentée ci-après :

// server.js

const express = require('express');

const bodyParser = require('body-parser');

const db = require('./models');

const app = express();

app.use(bodyParser.json());

app.use(express.static(\_\_dirname + '/static'));

app.get('/api/contacts', (req, res) => {

return db.Contact.findAll()

.then((contacts) => res.send(contacts))

.catch((err) => {

console.log('There was an error querying contacts', JSON.stringify(err))

return res.send(err)

});

});

app.post('/api/contacts', (req, res) => {

const { firstName, lastName, phone } = req.body

return db.Contact.create({ firstName, lastName, phone })

.then((contact) => res.send(contact))

.catch((err) => {

console.log('\*\*\*There was an error creating a contact', JSON.stringify(contact))

return res.status(400).send(err)

})

});

app.delete('/api/contacts/:id', (req, res) => {

const id = parseInt(req.params.id)

return db.Contact.findByPk(id)

.then((contact) => contact.destroy({ force: true }))

.then(() => res.send({ id }))

.catch((err) => {

console.log('\*\*\*Error deleting contact', JSON.stringify(err))

res.status(400).send(err)

})

});

app.put('/api/contacts/:id', (req, res) => {

const id = parseInt(req.params.id)

return db.Contact.findByPk(id)

.then((contact) => {

const { firstName, lastName, phone } = req.body

return contact.update({ firstName, lastName, phone })

.then(() => res.send(contact))

.catch((err) => {

console.log('\*\*\*Error updating contact', JSON.stringify(err))

res.status(400).send(err)

})

})

});

app.listen(3000, () => {

console.log('Server is up on port 3000');

});