

# Test Technique Fullstack

VIDAL Antoine

Qarnot, 4MTec

Février 2024



# Sommaire

- 1 Introduction
  - Présentation du projet
- 2 Architecture du projet
  - API REST
  - Bases de données
  - Application web
- 3 Implémentation de l'API REST
  - Modèle TypeAppareil
  - Routes
  - Schéma
- 4 Implémentation de l'application web
  - Composants Vuetify
  - Interface Utilisateur
- 5 Documentation et bonnes pratiques
- 6 Principaux problèmes rencontrés
- 7 Conclusion

# Introduction

## Présentation du projet

### Objectif

Développer une application permettant de manipuler une base de données via une API REST et de visualiser ces données à l'aide d'une application web.

L'architecture du projet repose sur deux technologies :

- **Node.js avec Express.js** : utilisé pour créer l'API REST qui permettra de manipuler la base de données.
- **Vue.js** : utilisé pour développer l'application web qui se servira de cette API REST pour afficher les données et permettre des interactions avec l'utilisateur.

La base de données quant à elle est **PostgreSQL** (avec comme ORM Sequelize).

# Architecture du projet

## API REST

Le dossier serveur contient le code de l'API REST développée avec Node.js et Express.js. Voici la structure de ce dossier :

- `config.ts` : Chargement des variables d'environnement.
- `database.ts` : Initialisation de la connexion à la base de données.
- `index.ts` : Point d'entrée de l'application.
- `middlewares/` : Contient les middlewares utilisés dans l'application, notamment `verifyToken.ts` pour la vérification des tokens JWT.
- `models/` : Modèles de données correspondant aux entités de la base de données (utilisateurs, appareils, etc.).
- `routes/` : Définitions des routes de l'API REST.
- `schemas/` : Schémas de validation des données utilisés pour valider les entrées de l'API.
- `utils/` : Utilitaires divers, comme `TokenUtils.ts` pour la gestion des tokens JWT.

Voici les modèles utilisés dans notre base de données PostgreSQL :

- **TypeAppareil** : Ce modèle représente les types d'appareils (sous forme de chaînes de caractères).
- **ModeleAppareil** : Ce modèle représente les modèles d'appareils, qui sont associés à un type d'appareil.
- **Appareil** : Ce modèle représente les appareils eux-mêmes, avec leur adresse MAC et leur état. Ils sont associés à un modèle d'appareil.
- **Connexion** : Ce modèle représente les connexions entre un appareil parent et un appareil enfant, avec date de début et de fin de cette connexion.
- **User** : Ce modèle représente les utilisateurs de l'application, avec leurs informations d'identification.

# Architecture du projet

## Application web

Le code source de l'application web se trouve dans le dossier `client`. Son contenu est le suivant :

- `routes.ts` : fichier contenant les routes de l'application Vue.
- `components/` : dossier contenant la liste des différents composants de l'application.
  - `FooterComponent.vue` : le footer de l'application.
  - `ListeAppareils.vue` : le composant principal de notre application permettant de visualiser les données.
  - `LoginPage.vue` : la page de login permettant d'accéder aux données et de manipuler l'API.
  - `NavBar.vue` : le header de notre application.
  - `NotFound.vue` : le composant à afficher lorsque la page demandée n'existe pas.
  - `SignUpPage.vue` : la page de création de compte pour obtenir l'accès à l'API.

# Implémentation de l'API REST

## Modèle TypeAppareil

```
1 import { DataTypes, Model, Sequelize } from 'sequelize';
2
3 You, last week | 1 author (You)
4 class TypeAppareil extends Model {
5   public idType!: number;
6   public nomType!: string;
7 }
8 export const initTypeAppareilModel = (sequelize: Sequelize) => {
9   TypeAppareil.init(
10     {
11       idType: {
12         type: DataTypes.INTEGER,
13         autoIncrement: true,
14         primaryKey: true,
15         field: 'id_type'
16       },
17       nomType: {
18         type: DataTypes.STRING(255),
19         allowNull: false,
20         unique: true,
21         field: 'nom_type'
22       },
23     },
24     {
25       sequelize,
26       tableName: 'type_appareils',
27       paranoid: true,
28     }
29   );
30 };
31
32 export default TypeAppareil;
```

Figure: Modèle d'un type d'appareil

# Implémentation de l'API REST

## Routes

- GET /type-appareils : Récupérer tous les types d'appareils. Cette route inclut des filtres pour retourner les données souhaitées.
- GET /type-appareils/:id : Récupérer un type d'appareil par son ID
- POST /type-appareils : Créer un nouveau type d'appareil
- PUT /type-appareils/:id : Mettre à jour un type d'appareil existant
- DELETE /type-appareils/:id : Supprimer un type d'appareil

Les routes POST et PUT effectuent des vérifications notamment pour vérifier l'existence d'un attribut. Ces vérifications ont été implémentées avec des validators.



# Implémentation de l'API REST

## Routes

```
44 router.get('/', async (req: Request, res: Response) => {
45   const { nomType } = req.query;
46   const optionsFiltre: any = {};
47
48   if (nomType) {
49     optionsFiltre.nomType = nomType;
50   }
51
52   try {
53     const types = await TypeAppareil.findAll({
54       where: optionsFiltre,
55     });
56
57     res.json(types);
58   } catch (error) {
59     console.error(error);
60     res.status(500).json({ error: 'Erreur interne au serveur' });
61   }
62 });
```

Figure: Route GET des types d'appareils

```
170 router.put('/:id', validateTypeAppareil, async (req: Request, res: Response) => {
171   const idType = req.params.id;
172   const { nomType } = req.body;
173
174   const typeExistant = await TypeAppareil.findOne({ where: { nomType } });
175
176   if (typeExistant) {
177     return res.status(409).json({ error: 'Nom de type déjà utilisé' });
178   }
179
180   try {
181     const type = await TypeAppareil.findById(idType);
182
183     if (!type) {
184       return res.status(404).json({ error: 'Type non trouvé' });
185     }
186
187     await type.update({ nomType });
188     res.json(type);
189   } catch (error) {
190     console.error(error);
191     res.status(500).json({ error: 'Erreur interne au serveur' });
192   }
193 });
```

Figure: Route PUT des types d'appareils

# Implémentation de l'API REST

## Schéma

```
1 import Joi from 'joi';
2
3 const typeAppareilSchema = Joi.object({
4   | nomType: Joi.string().max(255).required(),
5   | });
6
7 export { typeAppareilSchema };
```

Figure: Schéma de validation d'un type d'appareil

Les schémas ont été implémentés avec le package Joi.

# Implémentation de l'application web

## Composants Vuetify

Vuetify a été utilisé pour créer une interface utilisateur riche et interactive.  
Principaux composants utilisés :

- `v-data-table`: Pour afficher des données sous forme de tableau.
- `v-text-field`: Pour créer des champs de texte pour filtrer les données.
- `v-select`: Pour créer des menus déroulants.
- `v-icon`: Pour afficher des icônes dans l'interface utilisateur.

Axios a été utilisé pour utiliser l'API REST.

# Implémentation de l'application web

## Composants Vuetify

Pour afficher les données des connexions, nous utilisons le composant `v-data-table` de Vuetify. Voici un exemple de son utilisation :

```
<v-data-table :headers="connexionHeaders" :items="filtrerConnexions" :sort-by="['dateFin', 'dateDebut']"
:sort-desc="[true, true]" class="elevation-5">
```

Figure: Exemple d'utilisation du composant `v-data-table`

Nous utilisons également un filtre pour rechercher les appareils et connexions. Voici comment le filtre fonctionne :

```
filtre = filtre.filter((connexion) => {
  return connexion.appareilParent.adresseMAC.toLowerCase().startsWith(this.filtreMACParent.toLowerCase());
});
```

Figure: Exemple d'utilisation de notre filtre

# Implémentation de l'application web

## Interface Utilisateur

Liste des Appareils

Logout

Appareils











Modèle

Type

Adresse MAC

Filtrer par État  
tous

Nouvel Appareil

ID Appareil	Nom Modèle	Nom Type	Adresse MAC	État	Actions
1	Box OCP	Box	11:22:33:44:55:66	Installé	 
2	Radiateur 1700X	Radiateur	11:22:33:44:55:77	Installé	 
3	Chaudière OCP Leopard	Chaudière	11:22:33:44:55:88	Installé	 
4	Chaudière OCP Capri	Chaudière	11:22:33:44:55:99	maintenance	 
5	Radiateur 2700	Radiateur	11:22:33:44:55:10	stock	 
Rows per page: 10					1-5 of 5

Connexions

ID Appareil Enfant

Adresse MAC Enfant




ID Appareil Parent

Adresse MAC Parent

Date de Début

Date de Fin

Nouvelle Connexion

ID Connexion	ID Appareil Enfant	Adresse MAC Enfant	ID Appareil Parent	Adresse MAC Parent	Date de Début	Date de Fin	Actions
2	3	11:22:33:44:55:88	1	11:22:33:44:55:66	2022-12-01	2024-02-08	
1	2	11:22:33:44:55:77	1	11:22:33:44:55:66	2022-09-01	2024-02-08	
3	4	11:22:33:44:55:99	1	11:22:33:44:55:66	2022-09-01	2022-11-30	
Rows per page: 10					1-3 of 3		

2024 - AMTec

Figure: Aperçu de l'interface utilisateur

# Documentation et bonnes pratiques

- Utilisation du CamelCase
- Utilisation d'apidoc pour la documentation de l'API

```
15 /**
16  * @api {get} /type-appareils Récupérer tous les types d'appareils
17  * @apiVersion 0.1.0
18  * @apiName GetTypesAppareils
19  * @apiGroup TypeAppareil
20  *
21  * @apiHeader Authorization Bearer 'token JWT'. Token nécessaire à l'authentification.
22  *
23  * @apiParam {String} [nomType] Nom du type d'appareil à filtrer.
24  *
25  * @apiSuccess {Object[]} types Liste des types d'appareils.
26  * @apiSuccess {Number} idType ID du type d'appareil.
27  * @apiSuccess {String} nomType Nom du type d'appareil.
28  *
29  * @apiSuccessExample {json} Success-Response:
30  *   HTTP/1.1 200 OK
31  *   {
32  *     {
33  *       "idType": 1,
34  *       "nomType": "Box"
35  *     },
36  *     {
37  *       "idType": 2,
38  *       "nomType": "Radiateur"
39  *     }
40  *   ]
41  *
42  * @apiError {Error 500} {String} error Erreur interne au serveur.
43  */
```

Figure: Exemple de commentaire pour apidoc

# Principaux problèmes rencontrés

- Utilisation des tokens : stockage dans le LocalStorage
- Actualisation des données sans rafraîchir la page
- Meilleure vérification des adresses MAC (Joi ne supporte pas nativement les adresses MAC) : éviter de montrer une expression régulière à l'utilisateur
- Implémentation d'une date de fin optionnelle : choix d'utiliser "9999-12-31" et non pas une valeur indéfinie
- Utilisation de serveurs de production
- Utilisation du français et de l'anglais pour les erreurs

# Conclusion

- Résultat atteint : application Fullstack fonctionnelle avec une API REST pour manipuler les données
- Utilisation d'un ORM et de schémas côté Backend et de composants Vuetify côté Frontend pour faciliter le développement
- Améliorations possibles : utilisation d'un linter, ajout de tests (avec par exemple Jest), ou la séparation du composant ListeAppareils.vue en plusieurs fichiers pour une meilleure modularité.