Utiliser des validations et travailler avec une base de données existante avec Sequelize

De quoi s'agit ce Codelab ?

Dans le précédent Codelab, vous avez utilisé l'approche du code-first, dans laquelle vous créez les modèles avec NodeJS, puis Sequelize synchronise les modèles et crée-les dans la base de données. Dans ce Codelab, nous considérerons que notre base de données est déjà présente. Cela peut être le cas de la plupart des applications du monde réel que vous pouvez rencontrer, étant donné que nous commençons rarement un projet à partir de zéro, mais que nous commençons à travailler sur une application existante. Ce Codelab prendra en compte la base de données connue sous le nom de **chinook**.

base de données "chinook"

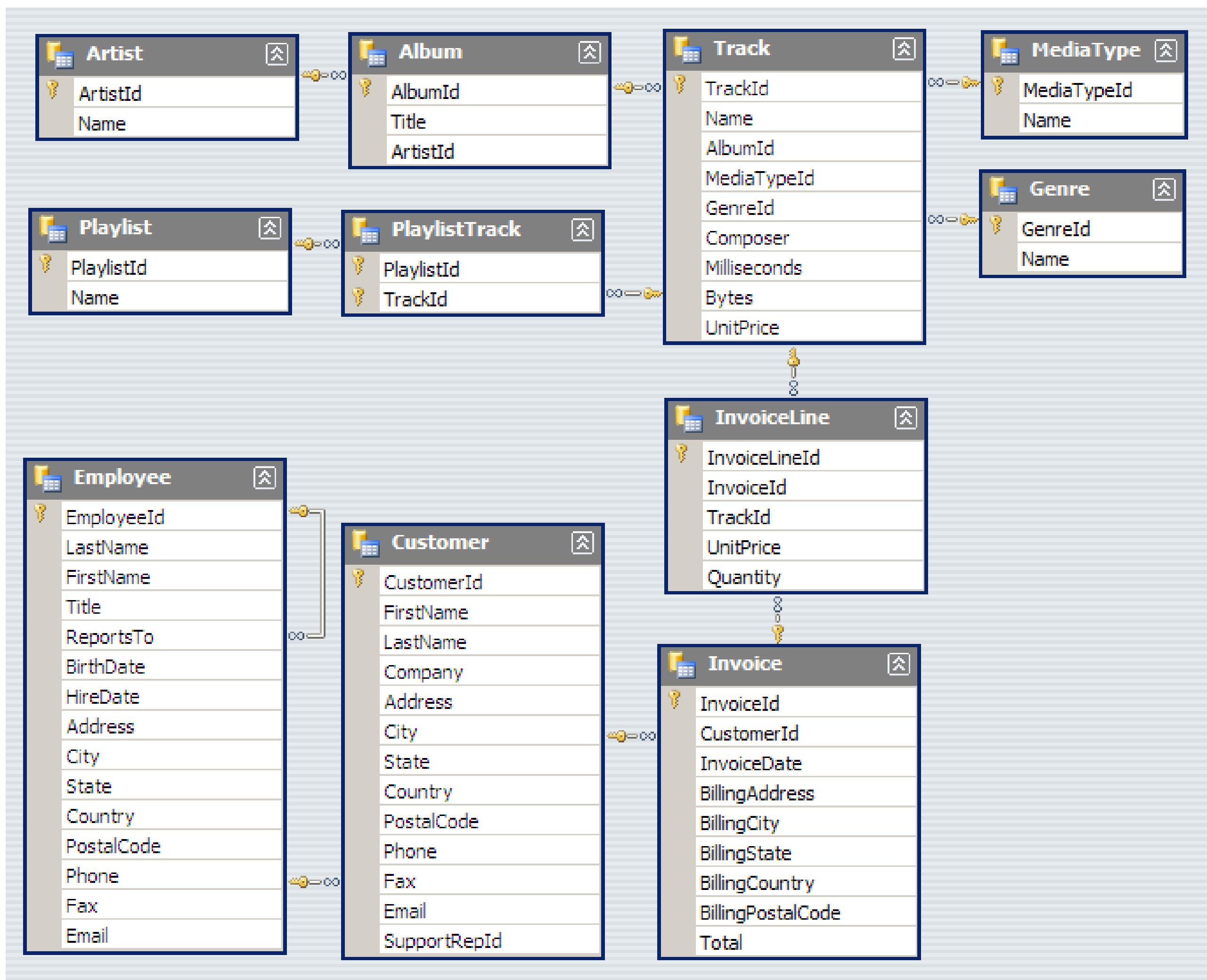
La base de données **chinook** est une bonne base de données pour s'entraîner avec SQL. Le diagramme de base de données suivant illustre les tables de la base de données et leurs relations.

Il y a 11 tables dans la base de données "**chinook**":

- La table des employés (**Employee)** stocke les données des employés telles que l'identifiant de l'employé, le nom, le prénom, etc. Elle comporte également un champ nommé **ReportsTo** pour spécifier qui relève de qui.
- La table des clients (**Customer**) stocke les données des clients.

- La tables des factures (**Invoice**) et des articles de facture (**InvoiceLine**) : ces deux tables stockent les données de facturation. La table des factures stocke les données d'en-tête de facture et la table des articles de facture stocke les données des éléments de ligne de facture.
- La table des artistes (**Artist**) stocke les données des artistes. Il s'agit d'un simple tableau qui ne contient que l'identifiant et le nom de l'artiste.
- La table des albums (**Album**) stocke des données sur une liste de morceaux de musique (**Track**). Chaque album appartient à un artiste. Cependant, un artiste peut avoir plusieurs albums.
- La table **MediaType** stocke les types de média tels que les fichiers audio MPEG et AAC.
- La table des genres (**Genre**) stocke les types de musique tels que le rock, le jazz, le métal, etc.
- La table **Track** stocke les données des chansons. Chaque morceau de musique appartient à un album.
- Les tables **Playlist** & **PlaylistTrack** : la table des playlists stocke des données sur les listes de lecture. Chaque liste de lecture contient une liste de chansons. Chaque

chanson peut appartenir à plusieurs listes de lecture. La relation entre la table **Playlist** et la table **Track** est **many-to-many**. La table **PlaylistTrack** est utilisée pour refléter cette relation.



Créez les tables et remplissez-les avec des données

Les scripts SQL pour créer et remplir les tables se trouvent sur cours-info:

- https://cours-info.iut-bm.univ-fcomte.fr/index.php/menu-cours-s3/r3-07-sql-en-langage-de-prog/2439-semaine-5-validations-et-contraintes-avec-sequelize

Utilisez **"chinook_ddl.sql"** pour créer les tables et **"chinook_seed.sql"** pour remplir la table avec des données. Si vous travaillez sur votre propre machine, vous pouvez créer une nouvelle base de données et la nommer "**chinook**" et l'utiliser pour ce Codelab.

1) Créer une base de données avec des paramètres par défaut

Tout d'abord, connectez-vous à PostgreSQL à l'aide de l'outil client **psql**. Exécutez l'instruction suivante dans une nouvelle base de données avec les paramètres par défaut :

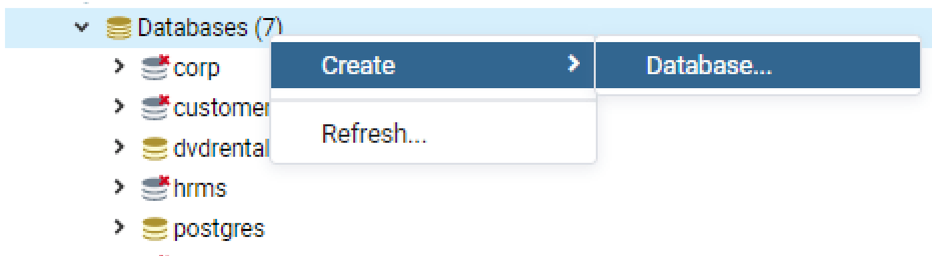
```
$ CREATE DATABASE chinook;
```

PostgreSQL créera une nouvelle base de données nommée "**chinook**" qui a des paramètres par défaut à partir de la base de données modèle par défaut (**template1**).
Si vous utilisez l'outil client **psql**, vous pouvez afficher toutes les bases de données du serveur de base de données PostgreSQL actuel à l'aide de la commande :

```
$ \l
```

2) Créer une nouvelle base de données à l'aide de pgAdmin

L'outil pgAdmin vous offre une interface intuitive pour créer une nouvelle base de données.
Tout d'abord, connectez-vous au serveur de base de données PostgreSQL à l'aide de **pgAdmin**.
Ensuite, cliquez avec le bouton droit sur le nœud **Databases** et sélectionnez l'élément de menu **Create** > **Database…** Il affichera une boîte de dialogue vous permettant d'entrer des informations détaillées sur la nouvelle base de données.



3) PostgreSQL sur les machines locales de l'IUT

Si vous travaillez sur une machine IUT locale, vous ne pourrez peut-être pas créer une nouvelle base de données. Dans ce cas, utilisez la base de données qui vous est attribuée "**baselogin**" et vous créez les tables directement dans cette base de données.

Vérifiez que les tables sont créées :

```
chinook=# \dt
              List of relations
 Schema |     Name      | Type  |
--------+---------------+-------+
 public | Album         | table |
 public | Artist        | table |
 public | Customer      | table |
 public | Employee      | table |
 public | Genre         | table |
 public | Invoice       | table |
 public | InvoiceLine   | table |
 public | MediaType     | table |
 public | Playlist      | table |
 public | PlaylistTrack | table |
 public | Track         | table |
```

Vérifiez que les tables sont remplies :

```
chinook=# select count(*) from "Album";
 count
-------
   347
(1 row)

chinook=# select count(*) from "Artist";
 count
-------
   275
(1 row)

chinook=# select count(*) from "Customer";
 count
-------
    59
(1 row)

chinook=# select count(*) from "Track";
 count
-------
  3503
(1 row)

chinook=# select count(*) from "Playlist";
 count
-------
    18
(1 row)
```

Créer un projet et se connecter à la base de données

Créez un nouveau projet NodeJS à l'aide de npm init dans un répertoire vide.

```
$ npm init
```

Ensuite, vous devrez installer les packages suivants :

```
$ npm install pg
$ npm install sequelize
$ npm install express body-parser
```

Pour utiliser Nodemon:

```
$ npm install —save-dev nodemon
```

Créez ensuite un dossier "**database**" et un fichier javascript à l'intérieur de ce dossier dans lequel vous vous connectez à la base de données via **Sequelize** et exportez l'objet "**Sequelize**" (référer au codelab précédent pour cette étape).

Créez ensuite le fichier d'entrée de votre application, importez express et lancez le serveur sur un port choisi.

Définir les modèles et associations

Créez un répertoire "**models**" qui contiendra vos modèles Sequelize.

```
$ mkdir models
```

Dans ce Codelab, nous ne créerons pas les modèles de toutes les tables, le but est de montrer comment nous pouvons travailler avec une base de données existante. Ainsi, nous allons considérer quelques tables.

Exercice 1 :

Dans votre fichier d'entrée javascript, créez une fonction **verifyDbCon** pour tester si la connexion est OK. Si l'authentification est ok, cette fonction imprime dans la console **"Connexion réussie!"**, sinon, elle imprime "**échec de la connexion**".

Assurez-vous que cette fonction affiche "Connexion réussie" avant de continuer dans ce codelab.

Créons maintenant notre premier modèle. N'oubliez pas que nous travaillons avec une base de données existante et que les tables sont déjà créées. Dans ce cas, lors de la création des modèles, nous devons définir le nom de la table existante dans le fichier à l'aide de l'option "**tableName**", et nous définissons l'option "**timestamps**" sur false pour ne pas ajouter de colonnes supplémentaires.

Commençons par le modèle d'album. Dans le répertoire "**models**", créez "**album.js**" et définissez le modèle comme suit :

```
const sequelize = require("chemin_fichier_config_bdd");
const { DataTypes } = require('sequelize');

module.exports = sequelize.define('album',{
    id: {
        field: 'AlbumId',
        type: DataTypes.INTEGER,
        primaryKey: true
    },
    name: {
        field: 'Title',
        type: DataTypes.STRING
    }
},{
    tableName: "Album",
    timestamps: false
});
```

Dans codelab, nous n'utiliserons pas le modèle routeur-contrôleur-service par souci de démonstration et de simplicité. Cependant, dans un projet réel, il n'est pas recommandé de mettre tout votre code dans votre fichier de point d'entrée (**server.js, index.js, app.js, etc**).

Ouvrez maintenant le fichier de point d'entrée javascript et importez le modèle **Album**. Ajoutez ensuite une fonction **syncModels()** qui synchronisera les modèles avec la base de données.

```
async function syncModels(){
    try{
        await Album.sync({force:false});
    } catch (error){
        console.error(error);
    }
}
```

Notez que dans le codelab précédent, nous avons synchronisé tous les modèles à la fois en utilisant "**await sequelize.sync({ force: true });**". Un modèle peut être synchronisé avec la base de données en appelant "**<model>.sync(options)**", une fonction asynchrone (qui renvoie une Promise). Avec cet appel, **Sequelize** effectuera automatiquement une requête SQL vers la base de données. Notez l'option **{force:false}** ici, nous ne voulons pas jouer avec la base de données existante ou supprimer/mettre à jour les tables par erreur.

Vous pouvez appeler la fonction **syncModels()** après **verifyDbCon()** dans votre fichier de point d'entrée.

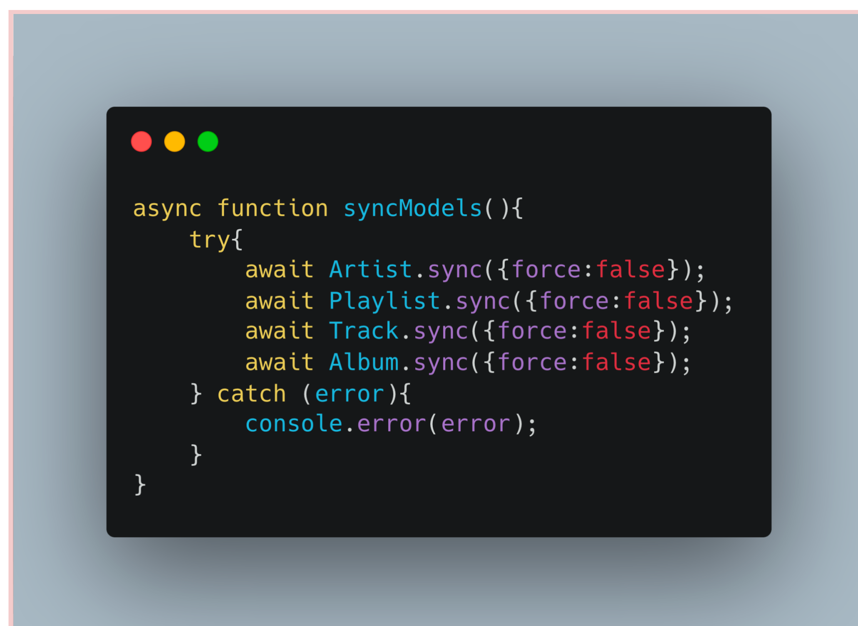
Exercice 2 :

Créez les modèles suivants :

- artist
  - ArtistId ⇒ autoincrement
- playlist
- track
  - Considérez uniquement les champs "**TrackId**" et "**Name**"

Importez les modèles dans le fichier d'entrée et définissez l'association entre l'artiste et l'album. Chaque album appartient à un artiste. Cependant, un artiste peut avoir plusieurs albums.

Mettez ensuite à jour la fonction **syncModels()** comme suit :



Maintenant, nous voulons associer le modèle "**playlist**" au modèle "**track**". Ici, nous avons une relation **many-to-many**. Nous devons d'abord ajouter un modèle qui représente la table de jointure. Créez un fichier **playlist.track.js** dans le répertoire “**models**” :

```
$ touch models/playlist.track.js
```

Définissez un modèle comme suit :

```
const sequelize = require("chemin_fichier_config_bdd");
const { DataTypes } = require('sequelize');

module.exports = sequelize.define('playlist_track',
    {},{
    tableName: "PlaylistTrack",
    timestamps: false
});
```

Une chose importante à noter ici est que nous n'avons pas défini les champs "**TrackId**" et "**PlaylistId**" dans le modèle de jointure. Cela devrait être fait lors de la définition des associations.
Dans votre fichier d'entrée, importez le modèle **PlaylistTrack** et vous pouvez définir les associations comme suit :

```
Playlist.belongsToMany(Track,{
   through: PlaylistTrack,
   foreignKey: 'PlaylistId',
});
Track.belongsToMany(Playlist,{
    through: PlaylistTrack,
    foreignKey: 'TrackId'
})
```

Mettez ensuite à jour la fonction **syncModels()** comme suit :

```
async function syncModels(){
    try{
        await Artist.sync({force:false});
        await Playlist.sync({force:false});
        await Track.sync({force:false});
        await PlaylistTrack.sync({force:false});
        await Album.sync({force:false});
    } catch (error){
        console.error(error);
    }
}
```

Ajouter une nouvelle table et des associations à une base de données existante

Disons que nous voulons ajouter une nouvelle table à la base de données existante en utilisant **Sequelize** directement. Cela est possible en créant un modèle et en utilisant l'option **{alter : true}** lors de la synchronisation avec la base de données.
Créons un modèle "**passport**" et attribuons-le à un artiste. La relation ici est **one-to-one**.

```
$ touch models/artist.passport.js
```

```javascript
const sequelize = require("chemin_fichier_config_bdd");
const { DataTypes } = require('sequelize');

module.exports = sequelize.define('passport',{
    id: {
        field: 'PassportId',
        type: DataTypes.UUID,
        primaryKey: true,
        defaultValue: DataTypes.UUIDV1
    },
    code: {
        field: 'Code',
        type: DataTypes.STRING
    },
    issueDate: {
        field: 'IssueDate',
        type: DataTypes.DATEONLY
    },
    expiryDate: {
        field: 'ExpiryDate',
        type: DataTypes.DATEONLY
    }
},{
    timestamps: false
});
```

Notez ici que nous n'avons pas utilisé l'option "**tableName**" car la table n'existe pas dans la base de données.

Importez le modèle de passeport dans le fichier d'entrée et associez-le au modèle de l'artiste comme suit :

```javascript
Artist.hasOne(Passport);
Passport.belongsTo(Artist);
```

Mettez ensuite à jour la fonction **syncModels()** comme suit :

```
async function syncModels(){
    try{
        await Passport.sync({alter:true});
        await Artist.sync({force:false, alter:true});
        await Playlist.sync({force:false});
        await Track.sync({force:false});
        await PlaylistTrack.sync({force:false});
        await Album.sync({force:false});
    } catch (error){
        console.error(error);
    }
}
```

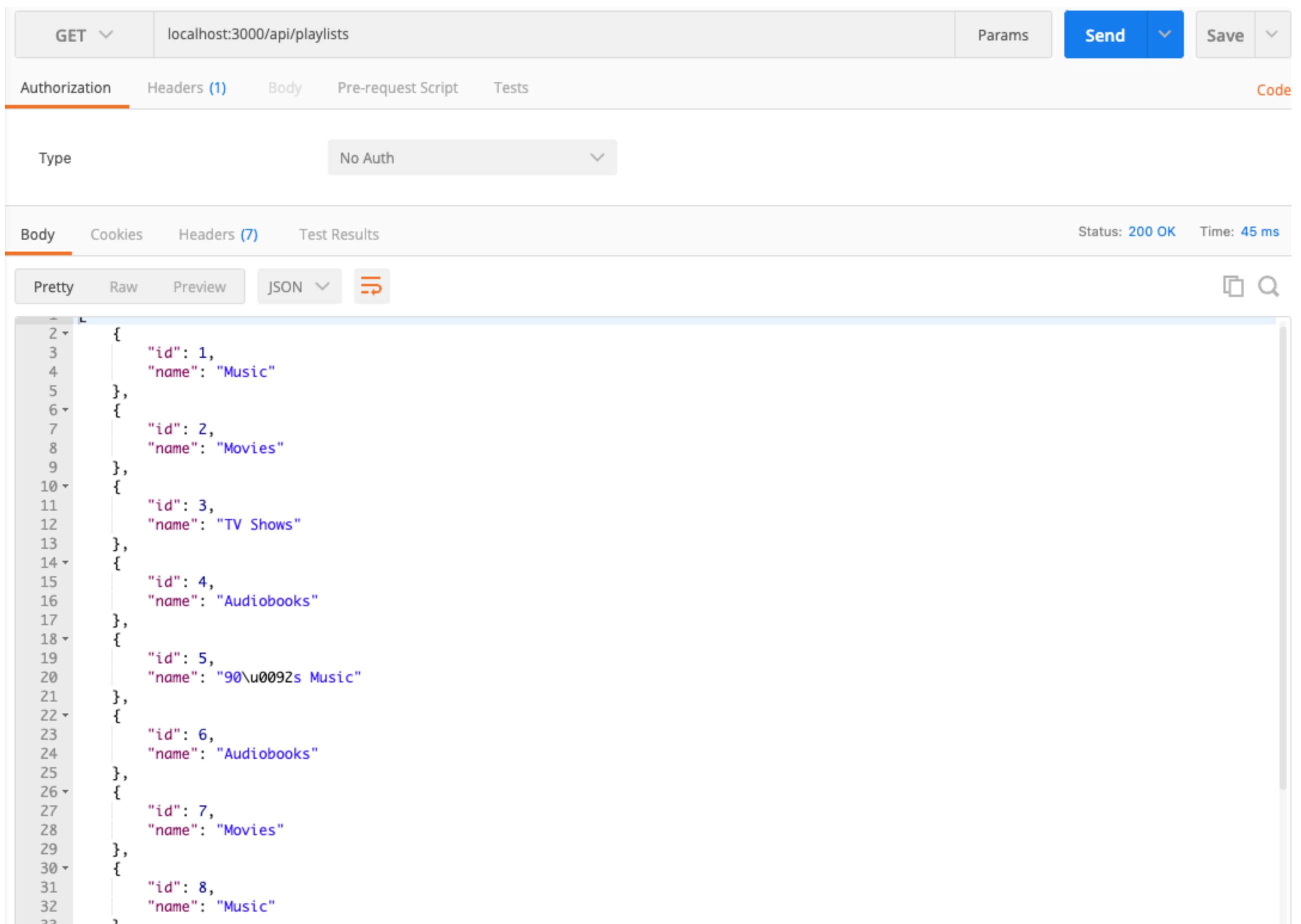
Une chose importante à noter ici est que nous avons ajouté l'option **{alter:true}** aux modèles **Passport** et **Artist** car nous voulons modifier ces deux modèles dans la base de données :

- Créer la table Passport si elle n'existe pas
- Associer l'artiste au passeport (ajout d'une référence de clé étrangère)

Créer des routes avec Express

Maintenant que nous avons créé les modèles et défini les associations, ajoutons quelques routes à notre application Web. Disons que nous voulons obtenir toutes les listes de lecture. Nous pouvons créer la route : **GET /api/playlists** et utiliser la méthode **findAll** comme suit :

```
app.get("/api/playlists",(req,res)=>{
    Playlist.findAll().then((playlists)=>{
        res.status(200).json(playlists);
    })
        .catch((e)=> {res.status(400).json("Not found!")});

});
```



Comme il existe souvent des cas d'utilisation dans lesquels il est simplement plus facile d'exécuter des requêtes SQL brutes / déjà préparées, vous pouvez utiliser la méthode **sequelize.query**. La requête ci-dessus est simple et ne nécessite pas d'écrire du SQL brut. Pour des raisons de démonstration, nous ferons la même requête en utilisant la fonctionnalité SQL brute dans **Sequelize**.

```
const { QueryTypes } = require('sequelize');

// ...
// ...
// ...
// ...

app.get("/api/playlists",(req,res)=>{
       sequelize.query(`SELECT * FROM "Playlist"`, { type: QueryTypes.SELECT })
        .then((playlists) => {
            res.status(200).send({success: 1, data: playlists});
        })
        .catch((e)=> {
            res.status(400).send({success: 0, data: e});
        })
});
```

Disons que nous voulons ajouter un filtrage au cas où une chaîne (query) **q** est fournie, nous pouvons le faire comme suit :

```
const { QueryTypes } = require('sequelize');

// ...
// ...
// ...
// ...

app.get("/api/playlists",(req,res)=>{
    sequelize.query(`SELECT * FROM "Playlist" WHERE "Name" LIKE :search_name`,
        {
            replacements: { search_name: `${req.query.q}%` },
            type: QueryTypes.SELECT
        })
        .then((playlists) => {
            res.status(200).send({success: 1, data: playlists});
        })
        .catch((e)=> {
            res.status(400).send({success: 0, data: e});
        })
});
```

GET localhost:3000/api/playlists?q=M Params Send Save

Body Cookies Headers (7) Test Results Status: 200 OK Time: 42 ms

Pretty Raw Preview JSON


```json
{
    "success": 1,
    "data": [
        {
            "PlaylistId": 1,
            "Name": "Music"
        },
        {
            "PlaylistId": 2,
            "Name": "Movies"
        },
        {
            "PlaylistId": 7,
            "Name": "Movies"
        },
        {
            "PlaylistId": 8,
            "Name": "Music"
        },
        {
            "PlaylistId": 9,
            "Name": "Music Videos"
        }
    ]
}
```


Exercice 3 :

Mettez à jour la route **GET /api/playlists** pour faire la même chose que ci-dessus mais cette fois en utilisant la méthode **findAll** et en y ajoutant le filtrage à l'aide d'opérateurs.

Vous devez importer et utiliser **Op**:

```
const {Op} = require('sequelize');
```

Vous pouvez vous référer à cette documentation :

https://sequelize.org/docs/v6/core-concepts/model-querying-basics/#operators

Exercice 4 :

Ajoutez la route GET **/api/playlists/id** pour trouver une playlist par clé primaire avec les "**tracks**" associés. L'identifiant doit être passé dans les params. Vous pouvez utiliser la méthode **findByPk**. *(ne pas utiliser sql brut)*

GET localhost:3000/api/playlists/8

Status: 200 OK Time: 122 ms

```
{
    "id": 8,
    "name": "Music",
    "tracks": [
        {
            "id": 1,
            "name": "For Those About To Rock (We Salute You)",
            "playlist_track": {
                "PlaylistId": 8,
                "TrackId": 1
            }
        },
        {
            "id": 2,
            "name": "Balls to the Wall",
            "playlist_track": {
                "PlaylistId": 8,
                "TrackId": 2
            }
        },
        {
            "id": 3,
            "name": "Fast As a Shark",
            "playlist_track": {
                "PlaylistId": 8,
                "TrackId": 3
            }
        },
```

Exercice 5 :

Ajoutez la route GET **/api/tracks/id** pour trouver un "track" par clé primaire avec les "playlists" associées. L'id doit être passé dans les params. *(ne pas utiliser sql brut)*

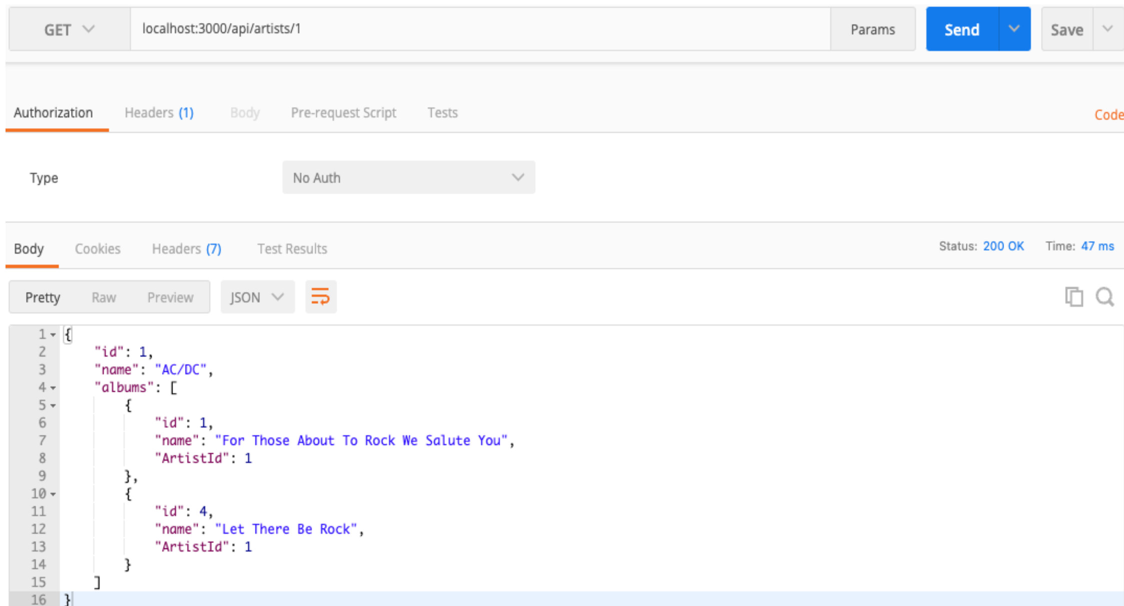
GET localhost:3000/api/tracks/8

Status: 200 OK Time: 82 ms

```
{
    "id": 8,
    "name": "Inject The Venom",
    "playlists": [
        {
            "id": 1,
            "name": "Music",
            "playlist_track": {
                "PlaylistId": 1,
                "TrackId": 8
            }
        },
        {
            "id": 8,
            "name": "Music",
            "playlist_track": {
                "PlaylistId": 8,
                "TrackId": 8
            }
        }
    ]
}
```

Exercice 6 :

Ajoutez la route GET **/api/artists/id** qui renvoie un artiste avec tous les albums associés. L'id doit être passé dans les params. *(ne pas utiliser sql brut)*



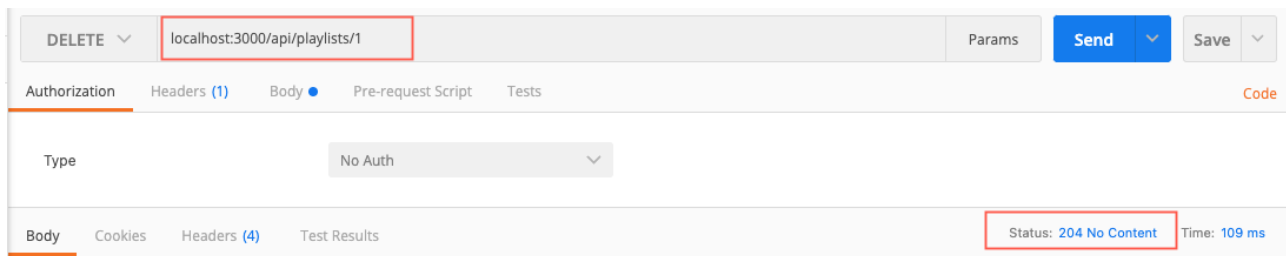
Exercice 7 :

Ajoutez la route GET **/api/albums/id** qui renvoie un album avec tous les artistes associés. L'id doit être passé dans les params. *(ne pas utiliser sql brut)*

Suppression d'une liste de lecture

Supprimer une “**playlist**” est une opération intéressante. La raison en est qu'une liste de lecture est associée à "**Track**" via la table de jointure "**PlaylistTrack**". Afin de supprimer une **playlist** avec un identifiant donné, nous devons d'abord supprimer les enregistrements de la table de jointure avec cet identifiant, puis supprimer la "**playlist**" de la table "**Playlist**".
Cela pourrait être fait comme suit :

```
// delete a playlist
app.delete('/api/playlists/:id',(req,res)=>{
   let { id } = req.params;
   Playlist
       .findByPk(id)
       .then((playlist)=>{
           if (playlist){
               // nous devrions supprimer tous les enregistrements
               // de la table de jointure PlaylistTrack
               return playlist.setTracks([]).then(()=>{
                   // après avoir supprimé la liste de lecture dans la table de jointure,
                   // nous pouvons supprimer de la table Playlist
                   return playlist.destroy();
               });
           }else {
               return Promise.reject();
           }
       })
       .then(() => {
           res.status(204).send(); // 204 No Content
       })
       .catch(()=> {
           res.status(404).send();
       })
});
```



Validation des modèles

Il est important de maintenir la cohérence et l'intégrité des bases de données. Les bases de données utilisent souvent une certaine forme de stipulation de contrainte pour assurer la cohérence. En règle générale, ces contraintes consistent à vérifier une plage de valeurs, telles que la longueur de chaîne minimale, l'unicité ou l'existence. L'intégrité des bases de données implique la gestion des associations et des relations entre les enregistrements. Cela implique des mises à jour et des suppressions en cascade des enregistrements référencés (par exemple, définir les colonnes d'identité associées sur NULL lorsque l'enregistrement référencé a été supprimé).

Le terme cohérence fait référence à la garantie que seules des données valides seront écrites et lues à partir de la base de données (en particulier dans le contexte de l'accès simultané aux données). L'intégrité fait référence aux données qui se conforment à un ensemble de règles et de contraintes avant d'être insérées ou lues.

Alors que la plupart des moteurs de base de données gèrent à la fois la cohérence et l'intégrité, il existe certaines limitations en ce qui concerne la cohérence. Si vous souhaitez effectuer des validations par rapport à une source tierce en dehors de la portée de la base de données, vous devez soit créer (ou installer) une extension pour la base de données

qui ajoute la prise en charge, soit utiliser une base de code centrale pour vous aider à gérer ces validations.

Sequelize propose une validation intégrée pour différents types de données afin d'aider à l'ergonomie d'un projet. Certaines validations nécessitent une configuration manuelle, telle que la vérification pour voir si une valeur de texte correspond à un modèle d'e-mail, ou une saisie manuelle pour certaines validations, telles que des plages numériques (ou de dates).

Les validations peuvent être effectuées à l'aide de deux méthodologies dans **Sequelize** :

- Nous pouvons exécuter des validations sur l'intégralité de l'enregistrement impliquant plusieurs attributs.
- Nous pouvons invoquer des validations pour chaque attribut spécifique.

Utilisation des validations comme contraintes

Il existe certaines validations que Sequelize utilisera à la fois comme validation et comme contrainte. Ces paramètres sont configurables dans les options de l'attribut comme frère des paramètres de validation. Les contraintes sont définies et protégées par la base de données, tandis qu'une validation sera gérée par **Sequelize** et le runtime **Node.js** exclusivement. Voici une liste de contraintes mises à disposition par Sequelize.

allowNull

L'option allowNull déterminera s'il faut appliquer NOT NULL aux définitions des colonnes de la base de données. La valeur par défaut est true, ce qui permettra aux colonnes d'avoir une valeur nulle. Il y a quelques mises en garde à garder à l'esprit lors de l'utilisation des validations avec la contrainte allowNull :

- Si le paramètre allowNull est défini sur false et que la valeur de l'attribut est nulle, les validations personnalisées ne s'exécuteront pas. Au lieu de cela, une ValidationError sera renvoyée sans faire de requête à la base de données.
- Si le paramètre allowNull est défini sur true et que la valeur de l'attribut est null, les validateurs intégrés ne seront pas invoqués, mais les validateurs personnalisés continueront de s'exécuter.

Pour illustrer ce qui précède, mettez à jour le modèle "**Artist**" comme suit :

- définissez **allowNull** sur **false** pour le champ "**name**" et ajoutez des validations intégrées.
- ajoutez un nouveau champ "**surname**" avec allowNull défini sur true, et ajoutez-y un validateur personnalisé.

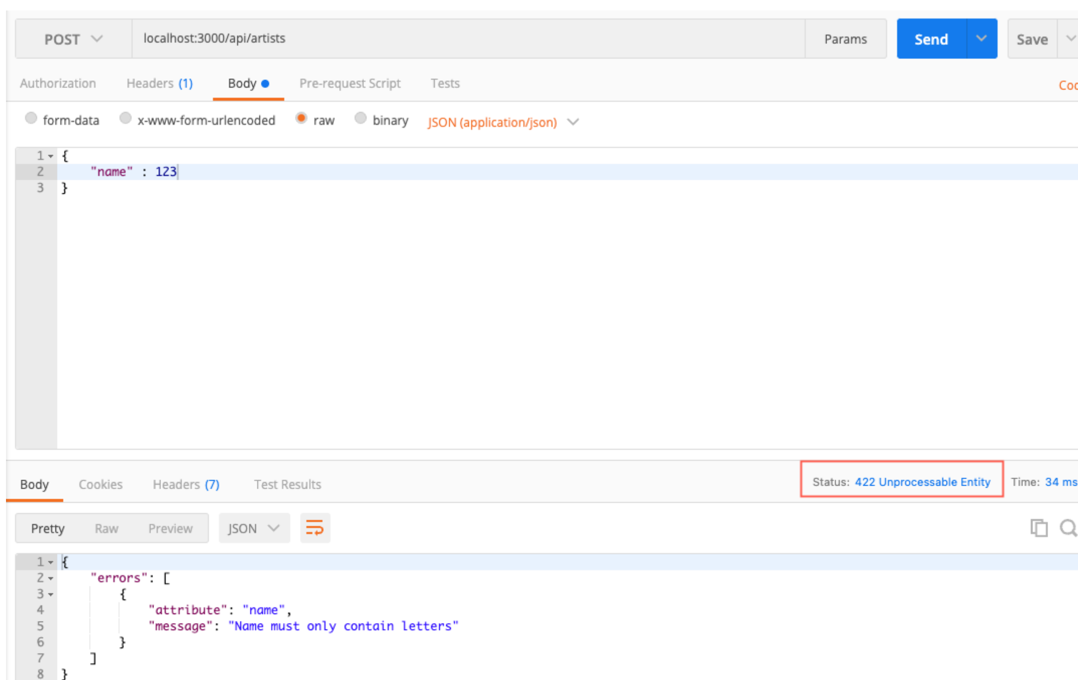
```
const sequelize = require("chemin_fichier_config_bdd");
const { DataTypes } = require('sequelize');

module.exports = sequelize.define('artist',{
    id: {
        field: 'ArtistId',
        type: DataTypes.INTEGER,
        primaryKey: true,
        autoIncrement: true
    },
    name: {
        field: 'Name',
        type: DataTypes.STRING,
        allowNull: false,
        // si la valeur du name est nulle, cela sera ignoré
        validate: {
            // Le nom ne doit pas être vide
            notEmpty: {
                args: true,
                msg: 'Name is required'
            },
            // Le nom ne doit contenir que des lettres
            isAlpha: {
                args: true,
                msg: 'Name must only contain letters'
            },
            // Le nom doit contenir entre 2 et 10 caractères
            len: {
                args: [2, 10],
                msg: 'Name must be between 2 and 10 characters'
            }
        }
    },
    surname : {
        field: 'Surname',
        type: DataTypes.STRING,
        allowNull: true,
        validate: {
            // même si la valeur du nom de famille est nulle,
            // le customValidator sera toujours invoqué
            customValidator(value){
                if(value === null && this.name.length < 3){
                    // Un nom de famille est requis sauf si l'artiste
                    // a un prénom avec plus de deux caractères
                    throw new Error("A surname is required unless the artist" +
                        " has a name with more than two characters");
                }
            }
        }
    }
},{
    tableName: "Artist",
    timestamps: false
});
```

Nous allons ajouter une route au point d'entrée de notre application pour créer un nouvel artiste et vérifier les contraintes et validations ajoutées :

```
app.post("/api/artists",(req,res)=>{
    let {  name, surname } = req.body;
    Artist.create({
        name: name,
        surname: surname
    }).then((artist)=>{
       res.json(artist);
    },(response)=>{
        res.status(422).json({errors:response.errors.map((error)=>{
                return {
                   attribute: error.path,
                   message: error.message
                }
            })
        });
    }).catch(err=> console.error(err));
});
```

Notez que la sortie de validation en cas d'erreur est accessible par le deuxième argument de rappel de la fonction "**then**". La méthode **then()** d'un objet **Promise** prend jusqu'à deux arguments : des fonctions de rappel pour les cas remplis et rejetés de la **Promise**. Notez que le deuxième argument (**response**) reflète une erreur dans la validation des données et non une erreur dans le code. Ce dernier est accessible dans la méthode "**catch**".

Testez la route :



POST localhost:3000/api/artists Params Send

Authorization Headers (1) Body Pre-request Script Tests

form-data x-www-form-urlencoded raw binary JSON (application/json)

```
{
    "surname" : ""
}
```

Body Cookies Headers (7) Test Results Status: 422 Unprocessable Entity

Pretty Raw Preview JSON

```
{
    "errors": [
        {
            "attribute": "name",
            "message": "artist.name cannot be null"
        }
    ]
}
```

POST localhost:3000/api/artists Params Send Save

Authorization Headers (1) Body Pre-request Script Tests Code

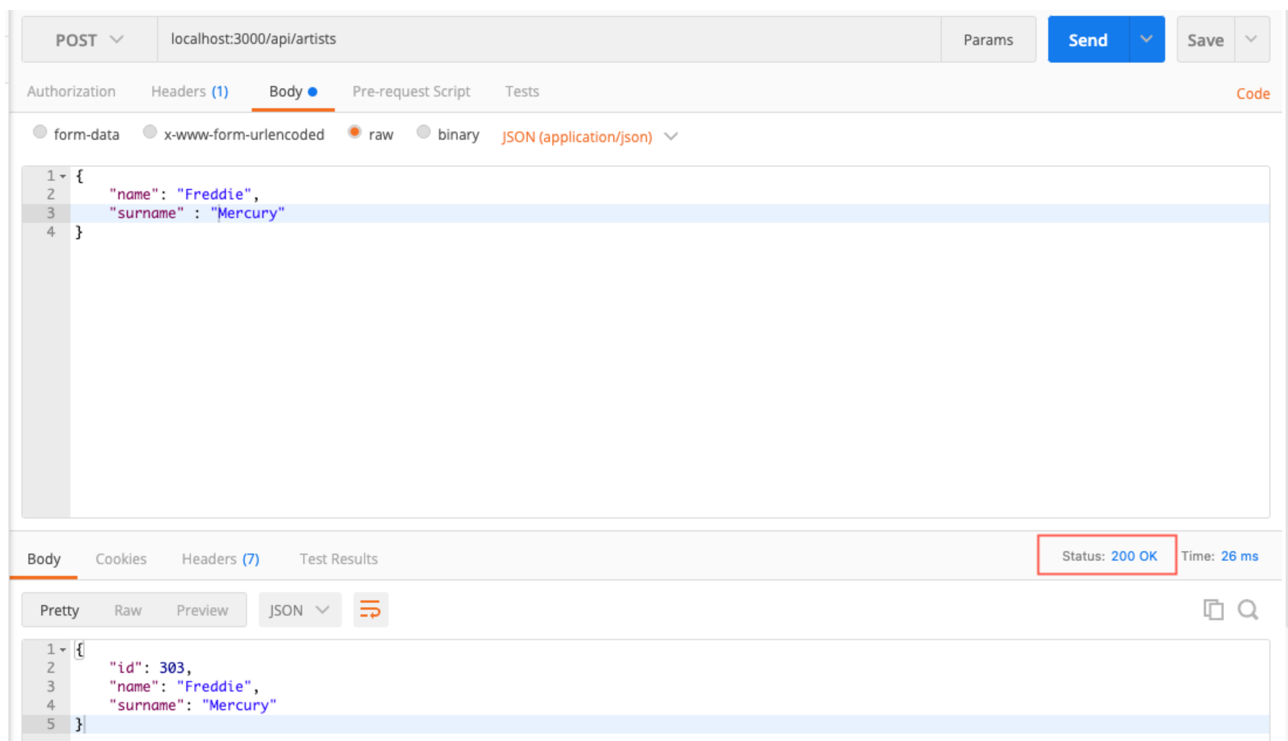
form-data x-www-form-urlencoded raw binary JSON (application/json)

```
{
    "name": "DJ",
    "surname" : null
}
```

Body Cookies Headers (7) Test Results Status: 422 Unprocessable Entity Time: 19 ms

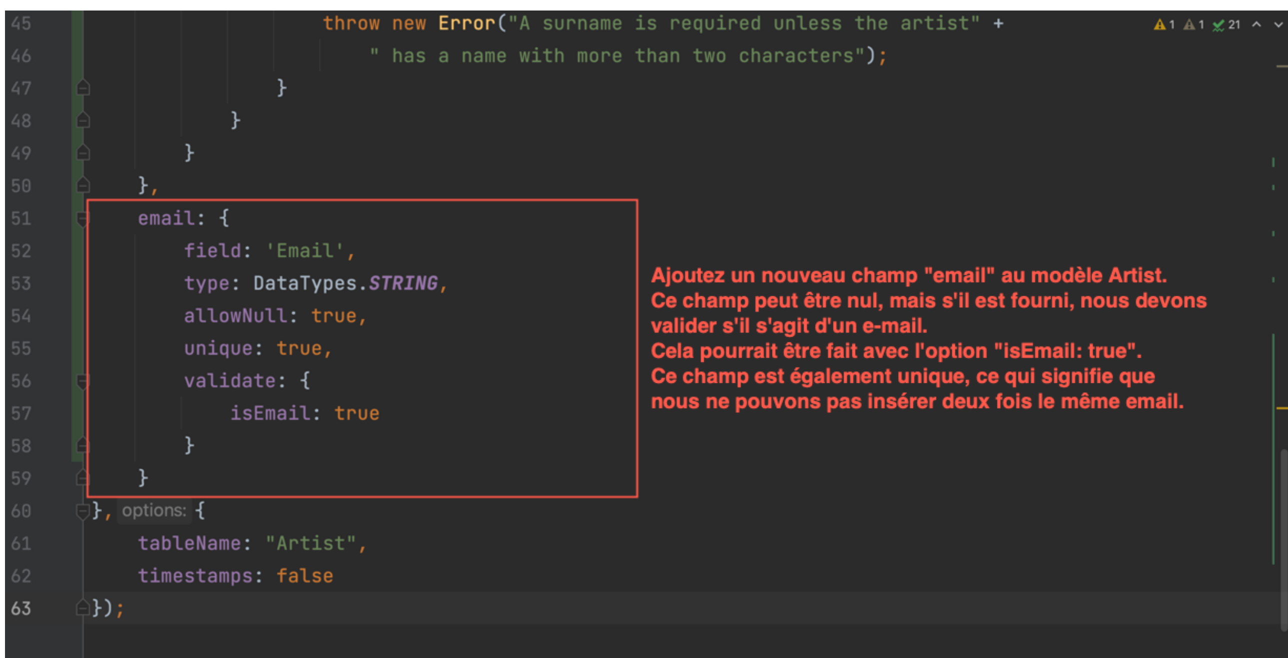
Pretty Raw Preview JSON

```
{
    "errors": [
        {
            "attribute": "surname",
            "message": "A surname is required unless the artist has a name with more than two characters"
        }
    ]
}
```



unique

Si vous définissez ce paramètre sur **true**, **Sequelize** créera une contrainte unique sur la colonne applicable dans la base de données si vous utilisez l'option de synchronisation de **Sequelize**. S'il y a eu une violation de contrainte unique, **Sequelize** renverra un type d'erreur de **SequelizeUniqueConstraintError**. Voici un exemple rapide de la façon d'utiliser **unique** (vous pouvez également autoriser des valeurs nulles pour l'unicité):



Mettez à jour la route **POST /api/artists** :

```
app.post("/api/artists",(req,res)=>{
    let {  name, surname, email } = req.body;
    Artist.create({
        name: name,
        surname: surname,
        email: email
    }).then((artist)=>{
        res.json(artist);
    },(response)=>{
        res.status(422).json({errors:response.errors.map((error)=>
{               return {
                    attribute: error.path,
                    message: error.message
                }
            })
        });
    }).catch(err=> console.error(err));
});
```

Testez votre route API :

POST localhost:3000/api/artists Params Send Save

Authorization Headers (1) Body Pre-request Script Tests Code

form-data x-www-form-urlencoded raw binary JSON (application/json)

```
{
    "name": "Freddie",
    "surname" : "Mercury",
    "email": "myEmail"
}
```

Body Cookies Headers (7) Test Results Status: 422 Unprocessable Entity Time: 44 ms

Pretty Raw Preview JSON

```
{
    "errors": [
        {
            "attribute": "email",
            "message": "Validation isEmail on email failed"
        }
    ]
}
```

POST localhost:3000/api/artists Params Send Save

Authorization Headers (1) Body Pre-request Script Tests Code

form-data x-www-form-urlencoded raw binary JSON (application/json)

```
{
    "name": "Freddie",
    "surname" : "Mercury",
    "email": "freddie.mercury@queen.com"
}
```

Body Cookies Headers (7) Test Results Status: 200 OK Time: 44 ms

Pretty Raw Preview JSON

```
{
    "id": 304,
    "name": "Freddie",
    "surname": "Mercury",
    "email": "freddie.mercury@queen.com"
}
```

Essayons de saisir à nouveau le même e-mail :

POST localhost:3000/api/artists Params Send Save

```
    "name": "Freddie",
    "surname" : "Mercury",
    "email": "freddie.mercury@queen.com"
}
```

Body Cookies Headers (7) Test Results Status: 422 Unprocessable Entity Time: 43 ms

Pretty Raw Preview JSON

```
{
    "errors": [
        {
            "attribute": "Email",
            "message": "Email must be unique"
        }
    ]
}
```

Essayons d'entrer une adresse e-mail nulle. Cela devrait être OK étant donné que nous autorisons les valeurs nulles pour le champ e-mail.

POST localhost:3000/api/artists Params Send Save

Authorization Headers (1) Body Pre-request Script Tests Code

form-data x-www-form-urlencoded raw binary JSON (application/json)

```
{
    "name": "Freddie",
    "surname" : "Mercury"
}
```

Body Cookies Headers (7) Test Results Status: 200 OK Time: 37 ms

Pretty Raw Preview JSON

```
{
    "id": 305,
    "name": "Freddie",
    "surname": "Mercury",
    "email": null
}
```

allowNull: true ==> donc c'est ok

Validations intégrées

Ces validations sont effectuées dans le runtime Node.js et non à partir de la base de données. **Sequelize** étendra les fonctionnalités de **validator.js** avec son propre ensemble de validateurs.

Les validateurs de modèle vous permettent de spécifier des validations de format/contenu/héritage pour chaque attribut du modèle. Les validations sont automatiquement exécutées lors de la création, de la mise à jour et de l'enregistrement. Vous pouvez également appeler **validate()** pour valider manuellement une instance. Vous pouvez définir vos validateurs personnalisés ou utiliser plusieurs validateurs intégrés, implémentés par **validator.js**, comme indiqué ci-dessous (capture de la documentation officielle) :

```
sequelize.define('foo', {
  bar: {
    type: DataTypes.STRING,
    validate: {
      is: /^[a-z]+$/i,          // matches this RegExp
      is: ["^[a-z]+$",'i'],     // same as above, but constructing the RegExp from a string
      not: /^[a-z]+$/i,         // does not match this RegExp
      not: ["^[a-z]+$",'i'],    // same as above, but constructing the RegExp from a string
      isEmail: true,            // checks for email format (foo@bar.com)
      isUrl: true,              // checks for url format (https://foo.com)
      isIP: true,               // checks for IPv4 (129.89.23.1) or IPv6 format
      isIPv4: true,             // checks for IPv4 (129.89.23.1)
      isIPv6: true,             // checks for IPv6 format
      isAlpha: true,            // will only allow letters
      isAlphanumeric: true,     // will only allow alphanumeric characters, so "_abc" will fail
      isNumeric: true,          // will only allow numbers
      isInt: true,              // checks for valid integers
      isFloat: true,            // checks for valid floating point numbers
      isDecimal: true,          // checks for any numbers
      isLowercase: true,        // checks for lowercase
      isUppercase: true,        // checks for uppercase
      notNull: true,            // won't allow null
      isNull: true,             // only allows null
      notEmpty: true,           // don't allow empty strings
      equals: 'specific value', // only allow a specific value
      contains: 'foo',          // force specific substrings
      notIn: [['foo', 'bar']],  // check the value is not one of these
      isIn: [['foo', 'bar']],   // check the value is one of these
      notContains: 'bar',       // don't allow specific substrings
      len: [2,10],              // only allow values with length between 2 and 10
      isUUID: 4,                // only allow uuids
      isDate: true,             // only allow date strings
      isAfter: "2011-11-05",    // only allow date strings after a specific date
      isBefore: "2011-11-05",   // only allow date strings before a specific date
      max: 23,                  // only allow values <= 23
      min: 23,                  // only allow values >= 23
      isCreditCard: true,       // check for valid credit card numbers

      // Examples of custom validators:
      isEven(value) {
        if (parseInt(value) % 2 !== 0) {
          throw new Error('Only even values are allowed!');
        }
      }
      isGreaterThanOtherField(value) {
        if (parseInt(value) <= parseInt(this.otherField)) {
          throw new Error('Bar must be greater than otherField.');
        }
      }
    }
  }
});
```

Nous pouvons tester certaines validations intégrées dans notre modèle d'artiste en nous assurant que le prénom de l'artiste **ne contient que** les lettres « abcde » et que son nom de famille **ne contient pas que** les lettres « abcde ». Nous pouvons utiliser les expressions régulières pour cette tâche.

Mettez à jour le modèle d'artiste comme suit :

```
11      name: {
12          field: 'Name',
13          type: DataTypes.STRING,
14          allowNull: false,
15          // si la valeur du name est nulle, cela sera ignoré
16          validate: {
17              // Le nom ne doit pas être vide
18              notEmpty: {
19                  args: true,
20                  msg: 'Name is required'
21              },
22              // Le nom ne doit contenir que des lettres
23              isAlpha: {
24                  args: true,
25                  msg: 'Name must only contain letters'
26              },
27              // Le nom doit contenir entre 2 et 10 caractères
28              len: {
29                  args: [2, 10],
30                  msg: 'Name must be between 2 and 10 characters'
31              },
32              is: /^[a-e]+$/i
33          }
34      },
35      surname : {
36          field: 'Surname',
37          type: DataTypes.STRING,
38          allowNull: true,
39          validate: {
40              // même si la valeur du nom de famille est nulle,
41              // le customValidator sera toujours invoqué
42              customValidator(value){
43                  if(value === null && this.name.length < 3){
44                      // Un nom de famille est requis sauf si l'artiste
45                      // a un prénom avec plus de deux caractères
46                      throw new Error("A surname is required unless the artist" +
47                          " has a name with more than two characters");
48                  }
49              },
50              not: /^[a-e]+$/i
51          }
52      },
```

En savoir plus sur les expressions régulières :

- https://www.w3schools.com/js/js_regexp.asp
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions

Testons la route de création avec un prénom qui contient la lettre "**f**", nous devrions obtenir une erreur :

POST localhost:3000/api/artists Params Send Save

```
{
  "name": "abcdf",
  "surname" : "efgh",
  "email": "test.email@test.com"
}
```

Body Cookies Headers (7) Test Results Status: 422 Unprocessable Entity Time: 28 ms

Pretty Raw Preview JSON

```
{
    "errors": [
        {
            "attribute": "name",
            "message": "Validation is on name failed"
        }
    ]
}
```

Testons la route de création avec un nom de famille qui ne contient que les lettres "**abcde**", nous devrions obtenir une erreur :

POST localhost:3000/api/artists Params Send Save

```
{
  "name": "abcd",
  "surname" : "abce",
  "email": "test.email@test.com"
}
```

Body Cookies Headers (7) Test Results Status: 422 Unprocessable Entity Time: 44 ms

Pretty Raw Preview JSON

```
{
    "errors": [
        {
            "attribute": "surname",
            "message": "Validation not on surname failed"
        }
    ]
}
```

Exercice 8 :

Ajoutez une validation au modèle de l'artiste pour vous assurer que l'adresse e-mail provient d'un domaine français (devrait se terminer par **.fr**).

Testez la route POST /api/artists

Exercice 9 :

Ajoutez un champ d'âge au modèle d'artiste et assurez-vous que :

- C'est une valeur numérique
- C'est entre 12 et 80
- peut être nul (contrainte)
- est impair (15, 17, 21, etc.)
- Exclure les valeurs [25, 27, 29, 31, 33]

Testez la route POST /api/artists

Exercice 10 :

Ajoutez un champ "password" au modèle de l'artiste et assurez-vous que nous ne pouvons pas insérer l'un de ces mots de passe :

- password
- 123456
- p@ssw0rd
- 098765

Testez la route POST /api/artists

Validations à l'échelle du modèle (Model-wide validations)

Des validations peuvent également être définies pour vérifier le modèle après les validateurs spécifiques au champ. En utilisant cela, vous pouvez, par exemple, vous assurer que ni la latitude ni la longitude ne sont définies ou les deux, et échouer si l'une mais pas l'autre est définie.

Les méthodes de validateur de modèle sont appelées avec le contexte de l'objet de modèle et sont considérées comme ayant échoué si elles renvoient une erreur, sinon elles réussissent.

Un exemple applicable à ce projet pourrait être la validation que le mot de passe d'un artiste est différent de son prénom. Nous pouvons faire cette validation au niveau du modèle.

Vous pouvez ajouter cette validation comme suit au modèle de l'artiste (après avoir terminé l'exercice 10) :

```
}, options: {
    tableName: "Artist",
    timestamps: false,
    validate: {
        namePassMatch(){
            if(this.name === this.password){
                throw new Error("Password cannot be your name!!!");
            }
        }
    }
});
```

Vous pouvez maintenant tester la route **POST /api/artists** :

POST localhost:3000/api/artists | Params | Send | Save

Authorization | Headers (1) | Body | Pre-request Script | Tests | Code

form-data | x-www-form-urlencoded | raw | binary | JSON (application/json)

```
{
  "name": "abcde",
  "surname" : "abcefgh",
  "email": "test.test@testsequelize.fr",
  "age" : 15,
  "password": "abcde"
}
```

Body | Cookies | Headers (7) | Test Results | Status: 422 Unprocessable Entity | Time: 60 ms

Pretty | Raw | Preview | JSON

```
{
  "errors": [
    {
      "attribute": "namePassMatch",
      "message": "Password cannot be your name!!!"
    }
  ]
}
```