CHARTE DU DEVELOPPEUR

MARS 2021

74 RUE FAIDHERBE, 58000 NEVERS



# SOMMAIRE

ENVIRONNEMENTS 1

BASES DE DONNÉES 1

CHARTE DE DEVELOPPEMENT 1

ARCHITECTURE DES PROJETS 1

GÉNÉRALITÉS 1

Alignement 1

Vertical 1

Horizontal 2

Nomination conventionnelle 2

Documents 2

Code 2

Indentation 3

Casse 3

Commentaires 3

Fichiers 3

Fonctions 4

Variables 4

Angular 5

Définition 5

Génération de fichiers 5

Component 5

Guard 5

Service 6

Module 6

Ajout 6

Suppression 6

Formalisme 6

HTML 7

SCSS 7

TypeScript 7

Fonctionnement 8

Debug 8

Tests 8

Build 9

NodeJS 9

Définition 9

Utilisation 10

Module 10

Ajout 10

Suppression 10

Formalisme 11

Configuration 11

Model 11

Route 14

Fonctionnement 15

Environnement 15

Tests 16

Build 18

SQL 18

Formalisme général 18

Formalisme interne 20

GITHUB – GESTION DES VERSIONS 21

INTRODUCTION 21

DOCUMENTATION 21

Rédaction des modes opératoires 21

Modes opératoires technique 21

Accès 21

Diffusion 21

Modes opératoires fonctionnels 22

Accès 22

Diffusion 22

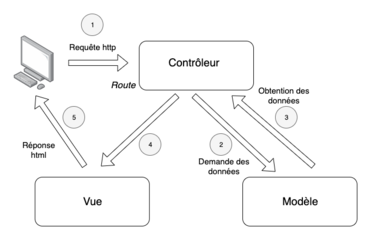
# ENVIRONNEMENTS

# BASES DE DONNÉES

# CHARTE DE DEVELOPPEMENT

## ARCHITECTURE DES PROJETS

L’ensemble des projets sont développés sous le principe du **MVC** (Modèle-Vue-Contrôleur).

**Modèle-vue-contrôleur** ou MVC est un motif d'architecture logicielle destiné aux interfaces graphiques lancé en 1978 et très populaire pour les applications web. Le motif est composé de trois types de modules ayant trois responsabilités différentes : les **modèles**, les **vues** et les **contrôleurs**.

Un **modèle** (Model) contient les **données à afficher**.

Une **vue** (View) contient la présentation de **l'interface graphique**.

Un **contrôleur** (Controller) contient la **logique** concernant les actions effectuées par l'utilisateur.

## GÉNÉRALITÉS

### Alignement

#### Vertical

Les **accolades** doivent respecter une **indentation simple** : **première** accolade **ouvrante** sur la même ligne que l’intitulé et **fermante** après un retour à la ligne.

***Exemples :***

recup\_utilisateurs() => {

}

if(x == 0) {

}

else {

}

**Déclarer** les **champs/propriétés** en **haut** de la **class** (public, private et protected).

#### Horizontal

Utiliser des **TAB** pour **l'indentation**.

Mettre un **espace** entre les **noms de variables** et les **mots clefs**.

Mettre un **espace** entre les **noms de variables** et les **opérateurs**.

**Séparer** les **paramètres** d'une **fonction** par des **espaces**.

**Ne pas** mettre **d'espace** entre les **noms** de **variable** et les **virgules**.

**Ne pas** mettre **d'espace** entre les **noms** de **variable** et les **parenthèses**.

**Éviter** les **lignes** de **plus** de **300 caractères**.

### Nomination conventionnelle

#### Documents

Les **documents** devront être **nommés** sous la forme **nom\_du\_document**, avec les **mots séparés** par des **underscore(\_),** le tout rangé dans le **dossier documentation du github.**

#### Code

Des **abréviations** de mots communs peuvent être employés du moment qu’elles restent **compréhensibles** par l’ensemble des personnes.

Les **noms suivants** sont des **exceptions** destinées à des **modules** ne représentant pas forcement une chose concrète, et n’ayant **pas besoin** d’un **nommage précis** de par leur **convention** dans la **communauté de développeur** :

• Itérateur dans une boucle ➔ i, j, …

• Paramètre d'une fonction native ➔ x, y, …

• Paramètre d'une fonction de bas niveau ➔ val, obj, err, …

### Indentation

Une **tabulation** correspondra toujours à un espacement équivalent à **2 espaces**.

C

### Casse

Il est **préférable** de **n'utiliser** des **majuscules** que pour **séparer** **deux mots** bien distincts. Dans ce cas la **première lettre** sera toujours en **minuscule**. Autrement, les **nominations** s’effectuent en suivant le **format** : **a\_b\_c** sauf pour la fonction où l’on veille à respecter le format impliquant la séparation des mots avec des majuscules. Cela permettant d’ores et déjà de faire une **distinction** au niveau de la **convention de nommage** entre une **fonction** et une **variable** par exemple.

### Commentaires

Dans un **ordre général** un **commentaire** peut être ajouté à **tout endroit du code** pour **expliquer** un **comportement** ou **apporter** des **informations supplémentaires** sur un **fonctionnement**.

#### Fichiers

Pour tout fichier, une **description succincte** de l’utilité de celui-ci devra être précisé **avant le code** en lui-même dans la mesure d’une nécessité relative évaluée par le développeur (inutile d’écrire un pavé pour un fichier de 3 lignes).

***Exemple :***

/\*

Fichier : utilisateur.component.ts

Description : Les components sont la composante principale d'Angular.

Chaque component contient :

• Un template HTML qui définit le rendu visuel de la page

• Une class Typescript qui sert à gérer le comportement du component

• Un fichier SCSS qui définit les modalités stylistiques du component

Le component utilisateur permet d'afficher la liste des utilisateurs et de définir les différentes actions en interaction avec la page

\*/

#### Fonctions

Pour une **fonction**, il faudra stipuler à quoi correspondent les **paramètres d’entrées** (**@params)** et **que retourne la fonction (@return)**, ainsi qu’une **brève description** de son **utilité.**

***Exemple :***

/\* Suppression d'un ticket

  @params : id\_ticket (explicite)

  @return : retourne une promesse avec l'id du ticket supprimé

  \*/

  supprimerTicket(id\_ticket: number): Promise<any[]> {

    return new Promise((resolve, reject) => {

      this.http.delete(environment.urlAPI+"/ticket/" + id\_ticket, {

        headers: {

          Authorization: 'Bearer ' + this.authentificationService.token,

        },

      })

      .subscribe((ticket : any) => {

        resolve(ticket);

      },

      (err) => {

        reject(err);

      });

    });

  }

Dans le cadre de **promesse** comme ci-dessus (**subscribe(), .then(), .catch(),** …) :

On effectuera les **retours à ligne** comme précisé dans l’exemple ci-dessus.

#### Variables

Un **commentaire** expliquant **l’utilité** d’une **variable** est nécessaire si son nom seul ne permet pas de lever **l’ambiguïté** entourant son **utilisation** ou si son **auteur** souhaite **apporter** des **précisions** sur son **utilisation**.

***Exemple :***

// Tableau des materiels selectionnés dans le selecteur multiple (<ng-multiselect-dropdown>)

  materiels\_selectionnes: any = [];

## Angular

### Définition

Développé par Google, Angular est un Framework open source écrit en JavaScript qui permet la création d’applications Web et plus particulièrement de ce qu’on appelle des « Single Page Applications » : des applications web accessibles via une page web unique qui permet de fluidifier l’expérience utilisateur et d’éviter les chargements de pages à chaque nouvelle action. Le Framework est basé sur une architecture du type MVC et permet donc de séparer les données, le visuel et les actions pour une meilleure gestion des responsabilités. Un type d’architecture qui a largement fait ses preuves et qui permet une forte maintenabilité et une amélioration du travail collaboratif.

### Génération de fichiers

#### Component

Les **components** doivent toujours être **nommé significativement** à leur **utilité.**

**Commande de génération : ng g c nomducomponent**

***Exemple : ng g c connexion***

#### Guard

Les **guards** permettent de **restreindre** notamment la **navigation** sur certaines pages aux **utilisateurs**. Ceux-ci doivent être **nommé significativement** et **placé** dans un **dossier guards** (src/guards).

**Commande de génération : ng g g nomduguard**

***Exemple : ng g g authentification***

#### Service

Un **service** permet d'avoir **accès** aux **données** récupérées en requêtant **l’API** depuis **n'importe quel endroit de l'application**. Ceux-ci doivent être **nommés significativement** et **placés** dans un **dossier services** (src/services).

**Commande de génération : ng g s nomduservice**

***Exemple : ng g s authentification***

### Module

Les **modules** sont ajoutés ou supprimés via le gestionnaire de paquet de NodeJS « **npm** ».

A

#### Ajout

Pour **ajouter** un **module**, on utilise la **commande** : **npm install nomdumodule –save**

#### Suppression

Pour **supprimer** un **module**, on utilise la commande : **npm uninstall nomdumodule –save-dev**

Cette commande supprimera également les dépendances du module. Une fois la suppression effectuée, si le module était utilisé quelque part alors la personne ayant désinstallé celui-ci sera chargé d’effectuer les modifications supplémentaires si nécessaires.

### Formalisme

**Utilisation** du Framework **bootstrap** pour la personnalisation visuelle des components.

#### HTML

Les **attributs** de type de **dénomination d’élément** dans des **balises** devront toujours commencer par un **préfixe** (minimum 3 lettres) équivalent au **type de balise** puis d’un underscore (\_) suivi d’un **suffixe** stipulant **l’élément du champ** (lui-même séparé par des underscore (\_) si celui-ci n’est pas composé d’un mot unique.

***Exemple :***

<input type="text" id="inp\_identifiant" class="form-control"

name="inp\_identifiant" ngModel required autofocus>

#### SCSS

Les **styles** seront toujours **rédigés** dans la **feuille** liée au **component**. La **nomination** des **éléments** se basent sur le **même** **principe** que pour le **HTML** :

***Exemple :***

.bouton\_deconnexion {

    position: fixed;

    bottom: 20px;

    right: 20px;

    width: 70px;

    height: 70px;

    padding: 10px 16px;

    border-radius: 35px;

    font-size: 12px;

    background-image: url("../../assets/bouton\_deconnexion.svg");

    background-repeat: no-repeat;

    background-size: 32px 32px;

    background-position: center;

    text-align: center;

}

#### TypeScript

En plus de **respecter** les **contraintes générales** définies précédemment, le **Typescript** se voit attribué un **typage** de ses **variables** assez spécifique. Dans notre cas lorsque le **type contenu** de la variable est **fixe** on définira alors le **type adéquat**. En revanche lors que le **type varie** tout autant que son contenu, alors on définira une variable de type « **any** » ou « **any []** » pour les tableaux.

***Exemples :***

utilisateurs: any = [];

id\_utilisateur: number

description\_ticket: string

### Fonctionnement

#### Debug

Pour lancer **l’application** en **mode** « **debug** » avec **le hot reloading** d’actif on utilise la commande : **ng s -o**

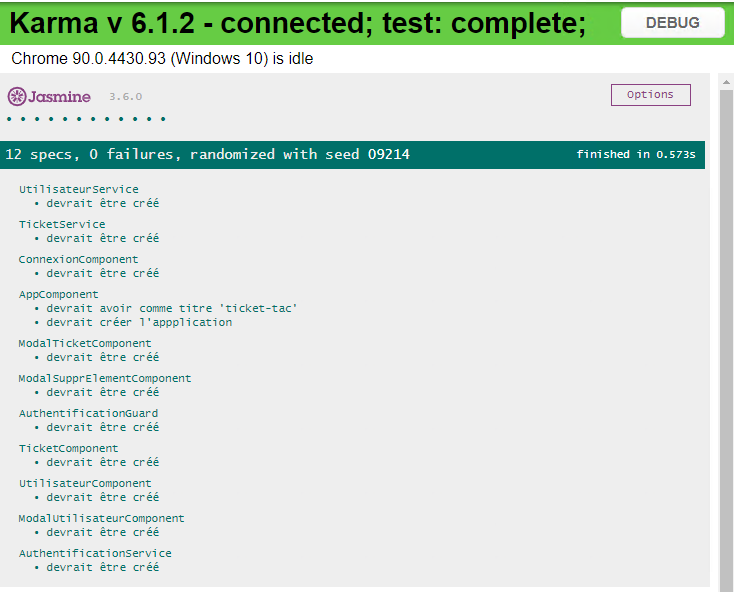
Cela ouvrira directement un onglet sur un navigateur pour aller sur la page initiale de l’application.

En cas **d’erreur de compilation** les erreurs sont **stipulées** sur la **page web** directement ainsi que dans la **console**.

#### Tests

Pour vérifier que **l’application** se **comporte** **normalement** on lance les **tests** de **l’application** avec la commande : **ng test**

Le **développeur** a pour charge **d’écrire** un **test** qui pour **chaque** **élément** vérifiera au minimum si celui-ci se **créer** et **fonctionne** correctement. Les **tests** sont **basés** sur le Framework **Jasmine.**



#### Build

Pour préparer **l’application au déploiement** dans les différents environnements, on utilise soit la commande : **ng build --configuration=dev**, pour le déploiement en développement, soit la commande : **ng build --configuration=preprod,** soit la commande **ng build --configuration=** **production**, pour le déploiement en production.

Cela va créer un dossier **dist** dans l’arborescence, c’est le contenu de ce dossier qui actera comme application.

## NodeJS

### Définition

Node.js est une plateforme logicielle libre en JavaScript, orientée vers les applications réseau événementielles hautement concurrentes qui doivent pouvoir monter en charge.

Elle utilise la machine virtuelle V8, la librairie libuv pour sa boucle d'évènements, et implémente sous licence MIT les spécifications CommonJS.

Parmi les modules natifs de Node.js, on retrouve http qui permet le développement de serveur HTTP. Il est donc possible de se passer de serveurs web tels que Nginx ou Apache lors du déploiement de sites et d'applications web développés avec Node.js.

Concrètement, Node.js est un environnement bas niveau permettant l’exécution de JavaScript côté serveur.

### Utilisation

Depuis plusieurs années, NodeJS, accompagné de son Framework Express, s'est fait une place dans le monde du développement web. Dans le même temps, le standard d'API REST s'est imposé comme référence pour les échanges de données entre serveurs et clients. La stack Node JS API REST est devenue un choix pertinent dans la conception de web services. C’est pourquoi nous l’utilisons.

### Module

Les **modules** sont ajoutés ou supprimés via le gestionnaire de paquet de NodeJS « **npm** ».

A

#### Ajout

Pour **ajouter** un **module**, on utilise la **commande** : **npm install nomdumodule –save**

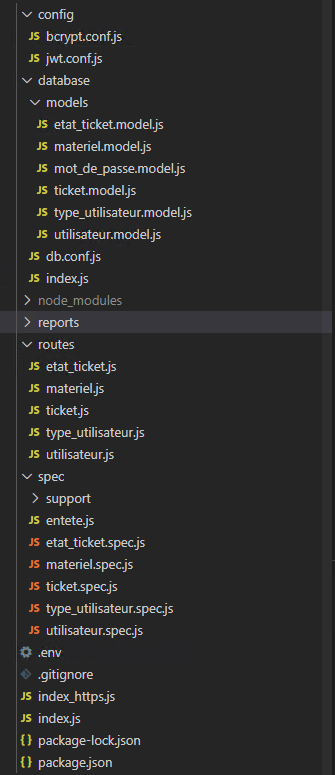
#### Suppression

Pour **supprimer** un **module**, on utilise la commande : **npm uninstall nomdumodule –save-dev**

Cette commande supprimera également les dépendances du module. Une fois la suppression effectuée, si le module était utilisé quelque part alors la personne ayant désinstallé celui-ci sera chargé d’effectuer les modifications supplémentaires si nécessaires.

### Formalisme

L’arborescence global de l’API se dessine de cette manière :



#### Configuration

Dans le dossier config, se trouve les différents **fichiers** de **configuration** nommé selon la norme **.conf.js** on y retrouve notamment les fichiers servant à la configuration du fonctionnement de l’authentification de l’utilisateur dans l’application Angular.



#### Model

Dans un premier temps, le fichier **db.conf.js** : C’est dans ce fichier qu’on **défini** les **modalités de connexion à la base de données** et aussi les différentes propriétés générales à prendre en compte pour **l’ORM** lors de la **génération** des **tables**.

// Connexion à la base de données et définition des propriétés par défaut pour les models

const sequelize = new Sequelize('ticket\_tac','postgres','postgres',{

    host: '127.0.0.1',

    dialect: 'postgres',

    define: {

        timestamps: false,

        freezeTableName: true,

        underscored: true

    }

});

On déclare ensuite dans le **sous dossier** **models** du **dossier database**, les **modèles** qui serviront à **l’ORM** pour **générer** nos **tables** dans la base de données.

Ci-dessous on peut voir qu’on ne déclare que l’id\_ticket en termes de champs avec une contraintes de clés étant donné que les liaisons entre les modèles se font dans un fichier différent (cela génèrera nos liaisons automatiquement).

// Définition du model ticket

module.exports = sequelize.define("ticket", {

    id\_ticket: {

        type: Sequelize.INTEGER,

        primaryKey: true,

        allowNull: false,

        autoIncrement: true

    },

    date\_resolution\_ticket: {

        type: Sequelize.DATE,

        allowNull: true

    },

    description\_ticket: {

        type: Sequelize.TEXT,

        allowNull: false

    },

    date\_saisie\_ticket: {

        type: Sequelize.DATE,

        allowNull: false,

        defaultValue: Sequelize.fn('NOW')

    }

});

Maintenant, le fichier index.js : C’est ici qu’on va **initialiser** la **synchronisation** avec la **base de données** et effectués les **liaisons** entre nos différents **modèles**.

/\*

Fichier : index.js

Description : Création des relations entre les models et définition de la fonction de synchronisation des données

\*/

// Import de sequelize et des différents models

const sequelize = require("./db.conf");

const ticket = require('./models/ticket.model.js');

const etat\_ticket = require('./models/etat\_ticket.model.js');

const utilisateur = require('./models/utilisateur.model.js');

const type\_utilisateur = require('./models/type\_utilisateur.model.js');

const materiel = require('./models/materiel.model.js');

const mot\_de\_passe = require('./models/mot\_de\_passe.model.js');

// Synchronisation des données avec les models

const Init = () => {

  sequelize.sync({ alter: true })

  .then(() => {

    console.log("Base de données synchronisée !")

  });

};

// Définition des relations entre les models

ticket.belongsTo(etat\_ticket, { foreignKey: 'fk\_etat\_ticket', as : 'etat\_ticket' });

ticket.belongsTo(utilisateur, { foreignKey: 'fk\_crea\_utilisateur', as: 'createur' });

ticket.belongsTo(utilisateur, { foreignKey: { name: 'fk\_modif\_utilisateur', allowNull: true } , as: 'modifieur' });

materiel.belongsToMany(ticket, { through: 'ticket\_materiel', foreignKey: 'fk\_materiel', otherKey: 'fk\_ticket' });

utilisateur.belongsTo(type\_utilisateur, { foreignKey: 'fk\_type\_utilisateur', onDelete: 'cascade', as: 'type\_utilisateur' });

utilisateur.belongsTo(mot\_de\_passe, { foreignKey: 'fk\_mot\_de\_passe', onDelete: 'cascade', as: 'mot\_de\_passe' });

// Export des modules

module.exports = {

  Init: Init,

  ticket: ticket,

  etat\_ticket: etat\_ticket,

  utilisateur: utilisateur,

  mot\_de\_passe: mot\_de\_passe,

  materiel: materiel,

  type\_utilisateur: type\_utilisateur

};

#### Route

Les différentes **routes** se trouvent dans le **dossier** **routes** de l’arborescence. On créer une route pour chaque type de données et on rédige les différentes routes pour ce type dans le même fichier. On utilisera également les **fonctionnalités** de **l’ORM** lorsque cela est **plus simple et plus clair** que d’écrire la requête **SQL** en elle-même. Chaque route se verra précédé d’un commentaire expliquant l’utilité de celle-ci.

***Exemple :***

/\*

Fichier : utilisateur.js

Description : Contient les routes de l'API pour toute action lié à un utilisateur

\*/

// Importation des modules

const express = require('express');

const db = require("../database/index");

const bcrypt = require("bcrypt");

const jwt = require("jsonwebtoken");

const router = express.Router();

const bcryptConf = require("../config/bcrypt.conf");

const sequelize = require("../database/db.conf");

const jwtConf = require("../config/jwt.conf");

// Récupération de tous les utilisateurs

router.get("/", (req, res) => {

    db.utilisateur.findAll({

        subQuery: false,

        order: [['id\_utilisateur', 'ASC']],

        include: [{

            model: db.type\_utilisateur,

            as: 'type\_utilisateur'

        }, {

            model: db.mot\_de\_passe,

            as: 'mot\_de\_passe'

        }],

        attributes: { exclude: ['fk\_mot\_de\_passe', 'fk\_type\_utilisateur'] }

    })

    .then(utilisateurs => {

        res.status(200).json(utilisateurs);

    })

    .catch(err => {

        res.status(500).json(err);

    });

});

// Suppression d'un utilisateur basé sur son id

router.delete("/:id", (req, res) => {

    sequelize.query(`DELETE FROM utilisateur WHERE id\_utilisateur = ` + req.params.id + ` RETURNING id\_utilisateur`

    , {

        // Affecte juste le format des données retournées, ici on retourne quelque chose donc on laisse QueryTypes.SELECT

        type: sequelize.QueryTypes.SELECT

    })

    .then(utilisateur => {

        if (utilisateur) {

            res.status(200).json("L'utilisateur " + utilisateur[0].identifiant\_utilisateur + " a été supprimé")

        }

        else {

            res.status(404).json();

        }

    })

    .catch(err => {

        res.status(500).json(err);

    });

});

### Fonctionnement

#### Environnement

Lors d’un **développement** **local**, les **configurations** de **l’API** se font avec les configurations par défaut de l’application : si le développeur souhaite modifier ses configurations il peut créer un **fichier .env** à la racine du dossier api pour configurer son projet, sous la forme suivante :

NODE\_ENV=nomdelenvironnement

PORT= numeroduport

BDD=nomdelabdd

UTILISATEUR=nomdelutilisateur

MOT\_DE\_PASSE=motdepasse

API\_URL= urldelapi

#### Tests

Pour lancer les tests on exécute la commande **npm test**. Tous les **tests** sont rédigés via le Framework **Jasmine** et sont classés dans un **dossier spec** avec une nomination par fichier équivalente au format nomspec.spec.js. Les **tests** doivent au minimum **vérifier** que les **routes** fonctionnent correctement en renvoyant le bon code de réponse http.

Le paquet utilisé pour faire des requêtes est « axios ».

Les **tests** se font de la manière suivante : il faut **écrire** toujours ses **tests** dans le **describe()** de chaque fichier, avec la fonction **it().**

A l'intérieur de cette dernière, on peut **tester plusieurs scénarios**. Prenons par exemple l'ajout, la modification et la suppression d'un ticket.

Tout d'abord, pour faire une **fonction**, il faut récupérer le **token** d’authentification pour pouvoir le passer dans les **headers** de chaque **requête**, ce qui se fait dans le **beforeEach()** de chaque fichier.

// test à faire avant chaque requête, pour s'authentifier

  beforeEach((done) => {

    axios.post(url\_base + ":" + port + "/utilisateur/connexion", entete)

    .then((reponse) => {

      expect(reponse.status).toEqual(200);

      token = reponse.data.token;

      done();

    })

    .catch((err) => {

      console.log(err);

    });

  });

Dans les fonctions **it()** et **describe(),** on doit **détailler** le plus possible le **scénario** du test fait et tester chacune des requêtes. On peut **commenter** légèrement le **code** si des calculs sont faits, mais le **describe()** et le **it()** doivent pouvoir suffire à la compréhension du code.



Si on doit écrire un **body** pour pouvoir tester des requêtes **post**, ils devront être **placés en haut du fichier, en dessous des imports.**

***Exemple :***

//test selectionne tout les etat\_ticket, retourne un code 200

  it("retourne code 200", (done) => {

    axios.get(url\_base + ":" + port + "/etat\_ticket", {

      headers: {

        'Authorization' : "Bearer " + token

      }

    })

    .then((reponse) => {

      expect(reponse.status).toEqual(200);

      done();

    })

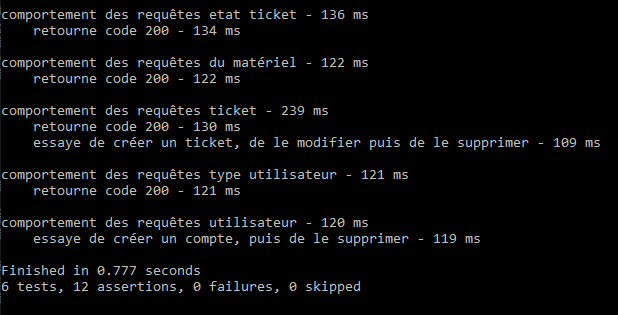
    .catch((err) => {

      console.log(err);

    });

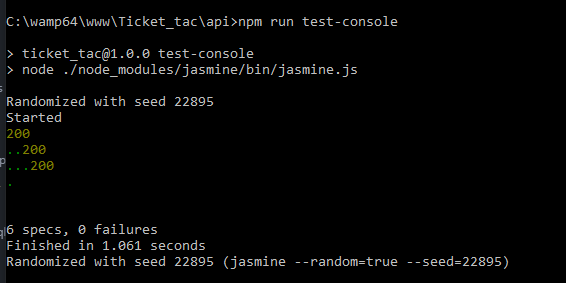
  });

Si dans le dossier **spec**, il n'y a pas de **dossier** **support**, il faudra lancer dans une invite de commande : **npm run test-init** dans le dossier api. Pour lancer les **tests**, lancer la commande : **npm test**. Il y a deux retours : un directement sur la console avec les temps d'exécution de chaque test.



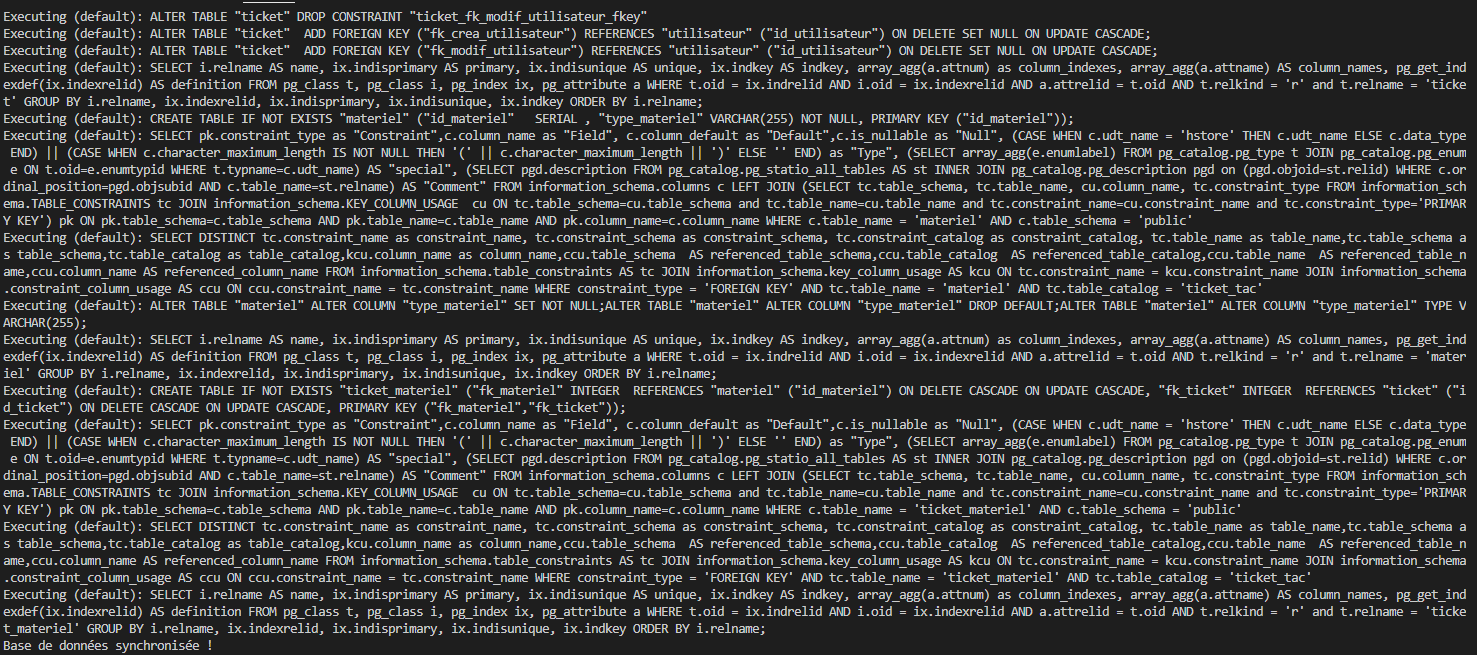
L’autre est un report automatique de junit en xml dans le dossier reports.

La deuxième commande est : **npm run test-console** : elle permet d'avoir le retour de la console en cas d'erreur.



#### Build

Pour **lancer l’application** il suffit **d’exécuter** la commande **node index.js** à la **racine** du **dossier** de **l’API.** Cela va lancer la construction des tables et/ou les synchroniser par rapport à nos modèles. Si le processus s’est déroulé sans accroc on obtient l’affichage suivant en console :



## SQL

### Formalisme général

Les **tables** doivent avoir une **dénomination** **claire** avec une **séparation** des mots via des **underscore** (\_).

Les **noms** de **colonne** doivent avoir leur appellation qui se **termine** par le **nom de la table** et chaque **mot** **séparé** par un **underscore** (\_).

Les **clefs** **étrangères** sont nommées **fk\_nomdelatable**.

Les **clefs primaires** doivent toutes être en **incrémentation automatique**.

Les **tables** **intermédiaires** sont nommées par la **concaténation des noms des tables** à partir desquelles elles sont créées.

***Exemples :***

***ticket\_materiel pour la table intermédiaire entre les tables ticket et matériel***

***description\_ticket pour la colonne description d’une table ticket***

***fk\_etat\_ticket pour la clef étrangère sur la table etat\_ticket***

Un retour à la ligne s’effectue à chaque utilisation de mots clef :

INSERT INTO ticket(descrition\_ticket, date\_saisie\_ticket, fk\_etat\_ticket, fk\_crea\_utilisateur)

VALUES (p\_description\_ticket, NOW(), 1, p\_id\_crea\_utilisateur)

Des **fonctions** en **langage** **PL/pgSQL** peuvent être écrite pour une succession de tâche plutôt que d’écrire une succession de requête dans l’API.

Le **nommage** de celle-ci se fait de la manière suivante : **func\_description\_de\_laction**.

Les **paramètres** seront toujours écrits avec le **préfixe** **p\_**

Les **variables** seront toujours écrites avec le **préfixe** **var\_**

Des **alias** peuvent être utilisés, ceux-ci correspondent soit à la **première lettre de la table** soit les **N premières** s’il y a une **ambiguïté**, voir à une **abréviation**.

On essaye de **revenir à la ligne** lorsque les paramètres d’une requête prennent plus de **200 caractères**.

***Exemple pour une fonction de création d’un ticket :***

CREATE OR REPLACE FUNCTION FUNC\_CREA\_TICKET(p\_id\_crea\_utilisateur INTEGER, p\_description\_ticket TEXT, p\_array\_id\_materiel INTEGER[])

RETURNS TABLE (id\_ticket INTEGER, description\_ticket TEXT, date\_saisie\_ticket TIMESTAMP WITH TIME ZONE, libelle\_etat\_ticket VARCHAR, identifiant\_utilisateur VARCHAR, type\_materiel VARCHAR)

AS

$$

DECLARE

  i INTEGER;

  var\_id\_ticket INTEGER;

BEGIN

  INSERT INTO ticket(description\_ticket, date\_saisie\_ticket, fk\_etat\_ticket, fk\_crea\_utilisateur)

  VALUES (p\_description\_ticket, NOW(), 1, p\_id\_crea\_utilisateur)

  RETURNING ticket.id\_ticket INTO var\_id\_ticket;

  FOREACH i IN ARRAY p\_array\_id\_materiel

  LOOP

    INSERT INTO ticket\_materiel(fk\_materiel, fk\_ticket)

    VALUES(i, var\_id\_ticket);

  END LOOP;

  RETURN QUERY SELECT DISTINCT

       t.id\_ticket

      ,t.description\_ticket

      ,t.date\_saisie\_ticket

      ,et.libelle\_etat\_ticket

      ,u.identifiant\_utilisateur

      ,m.type\_materiel

  FROM ticket t

  INNER JOIN etat\_ticket et ON et.id\_etat\_ticket = t.fk\_etat\_ticket

  INNER JOIN utilisateur u ON u.id\_utilisateur = t.fk\_crea\_utilisateur

  INNER JOIN ticket\_materiel tm ON tm.fk\_ticket = t.id\_ticket

  INNER JOIN materiel m ON m.id\_materiel = tm.fk\_materiel

  WHERE t.id\_ticket = var\_id\_ticket;

END;

$$

LANGUAGE PLPGSQL

### Formalisme interne

Pour ce qui est du **formalisme** à l’intérieur de **l’API**, comme stipulé précédemment on évite d’écrire les **requêtes** si l’on peut **privilégier** **l’ORM**, mais autrement on applique les **règles stipulées dans la section formalise** **générale** ci-dessus.

***Exemple :***



# GITHUB – GESTION DES VERSIONS

## INTRODUCTION

## DOCUMENTATION

### Rédaction des modes opératoires

### Modes opératoires technique

#### Accès

#### Diffusion

### Modes opératoires fonctionnels

#### Accès

#### Diffusion