Xabi **AVELLAN**
Sofian **EON**
Dylan **MENDIBOURE**
Arthur **MURILLO**

NOTEO

Manuel de maintenance

Groupe n°7

*Tuteur et commanditaire : Patrick **ETCHEVERRY***

- Projet tutoré semestre 4 | 2019 / 2020 -

Sommaire

Index des figures

Index des listings

1. Préambule

Pour comprendre ce manuel et les informations qui y sont renseignées, une connaissance minimale du Framework Symfony et de son fonctionnement est nécessaire. De plus, des notions d'algorithmiques y sont développées. Ce manuel s'adresse donc à des développeurs initiés.

2. Organisation du code

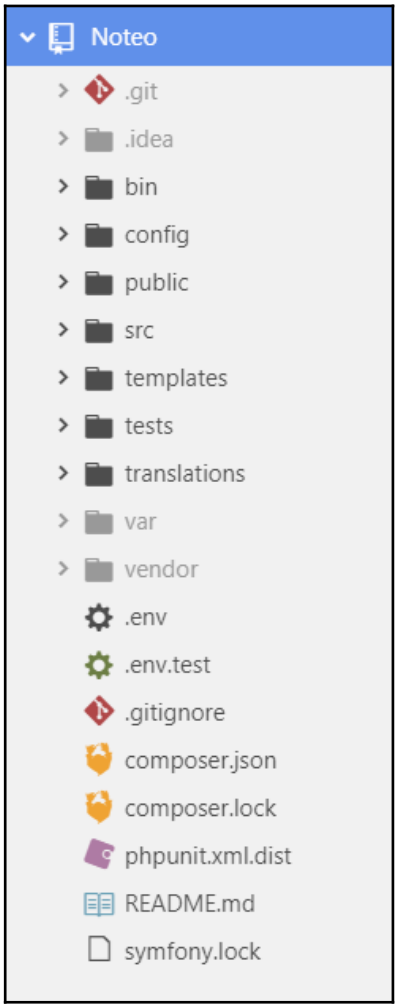

***Figure 1**: Arborescence principale du code*

La base de l'arborescence (***Figure 1***) est organisée en différents dossiers que nous présenterons, et en fichiers. Le fichier .env constitue le lien entre l'application et la base de donnée ou le serveur de mail par exemple. Il contient les identifiants permettant la connexion. Il est également possible de créer un fichier .env.local qui contiendra les informations propres au serveur et donc permet une plus grande

confidentialité car le fichier .env est poussé sur Github. Les fichiers composer.lock et .json contiennent les informations sur les différentes libraires, extensions, qui devront être installées par composer lors de la création de l'application. On retrouve également des ajouts nécessaires pour un dépôt git comme le fichier .git, .gitignore ou README.md.

Les différents dossiers sont les suivants :

2.1. Manipuler l'application

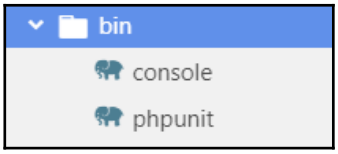

***Figure 2**: Dossier bin*

Le dossier bin (***Figure 2***) contient des scripts PHP exécutables permettant de lancer des tests unitaires (phpunit) ou d'exécuter des fonctions pour gérer l'application comme créer des entités, des formulaires, faire les migrations, lancer le serveur, … (console)

2.2. Gérer la configuration de l'application

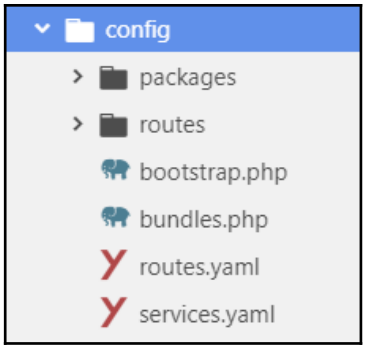

***Figure 3**: Dossier config*

Le dossier config (***Figure 3***) contient les différents fichiers de configuration de l'application. On y retrouvera par exemple le fichier de configuration de l'aspect sécurité de l'application (packages/security.yaml), ou de définition des différentes routes (routes.yaml). On y retrouve aussi les fichier des extensions installées.

2.3. Modifier les ressources de l'application

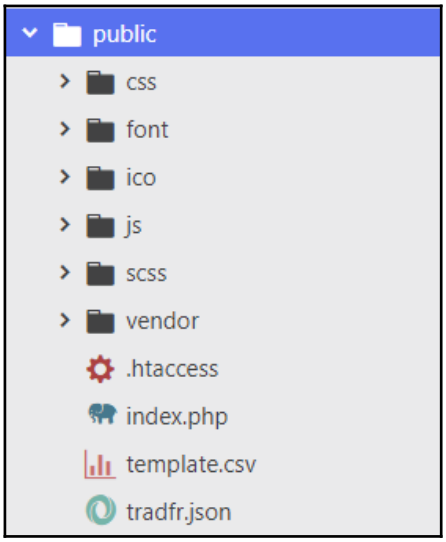

***Figure 4**: Dossier public*

Le dossier public (***Figure 4***) contient toutes les ressources dont le client accédant au serveur pourrait avoir besoin. On y retrouvera par exemple les fichiers css personnalisés, ou dans le cas de Noteo, les fichiers du template boostrap utilisé, ou le .htaccess régissant l'accès au serveur via le nom de domaine, ou la redirection https. S'y trouvent aussi les fichiers téléchargeables via l'application comme dans le cas de Noteo le modèle de fichier .csv pour l'importation d'étudiants.

2.4. Modifier le code source de l'application

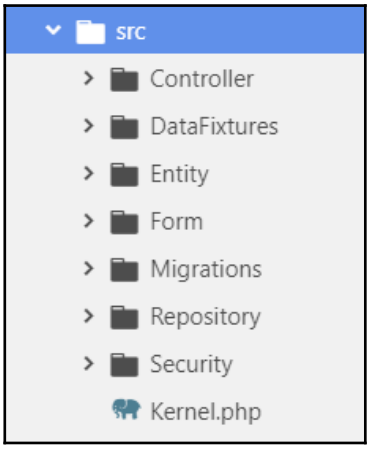

***Figure 5**: Dossier src*

Le dossier src (***Figure 5***) contient le code source de l'application. Dans l'architecture MVC, ce dossier contient le Modèle et le contrôleur. On y retrouve plusieurs sous-dossiers que nous allons présenter :

2.4.1. Modifier les fonctionnalités de l'application

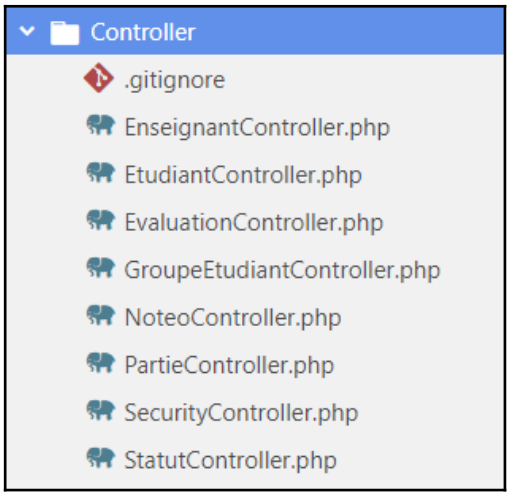

***Figure 6**: Dossier Controller*

Le dossier Controller (***Figure 6***) contient le code des différents contrôleurs utilisés dans l'application. Dans le cas de Noteo, on retrouve un contrôleur par entité. Chaque contrôleur gère pour l'entité les aspects du CRUD : Create (créer), Read (consulter), Update (modifier), Delete (supprimer). Pour le contrôleur d'évaluation on retrouve en plus l'aspect de gestion des statistiques que nous expliquerons dans la partie dédiée à certaines parties du code.

2.4.2. Modifier les données de test de l'application

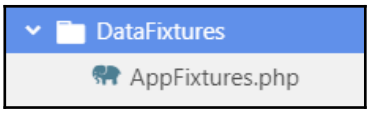

***Figure 7**: Dossier DataFixtures*

Le dossier DataFixtures (***Figure 7***) contient le code de la classe AppFixtures qui permet via une commande de charger dans la base de données des données de test que nous avons prédéfinies.

2.4.3. Modifier l'architecture de l'application

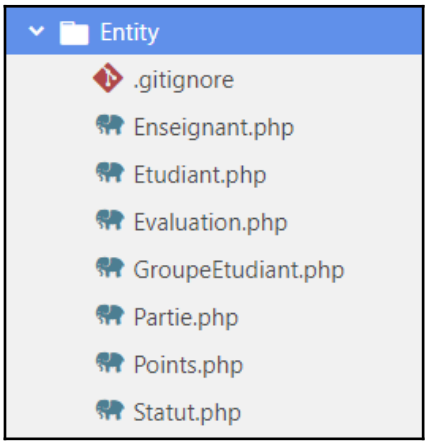

***Figure 8**: Dossier Entity*

Le dossier Entity (***Figure 8***) contient le code des entités (ou classes en notation UML par exemple) de l'application. On y retrouve donc pour chacune ses attributs, relations, accesseurs et mutateurs. On y retrouvera aussi, pour chaque attribut de chaque entité, des règles de validation pour définir si l'attribut est de la taille, du type, ou de la forme voulue.

2.4.4. Modifier les formulaires de l'application

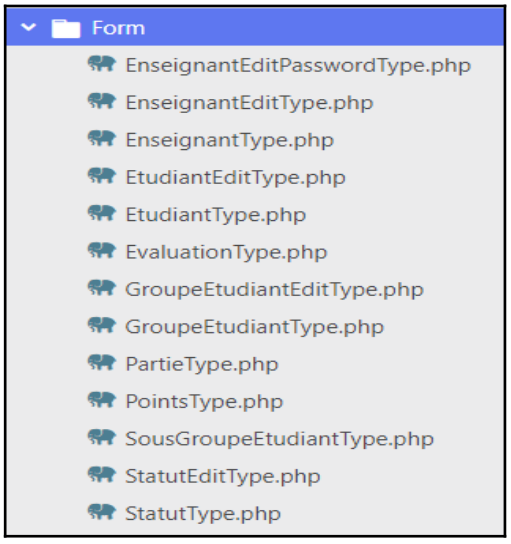

***Figure 9**: Dossier Form*

Le dossier Form (***Figure 9***) contient les différents formulaires utilisés par l'application. Le format utilisé est le Form Builder intégré dans Symfony. Tous ces formulaires portent sur des entités et permettent d'hydrater l'entité correspondante. Cependant on retrouve également une définition de formulaire dans le contrôleur de l'entité Évaluation car celui-ci n'est pas basé sur une entité précise, et que nous en avons besoin pour une action métier précise.

2.4.5. Modifier la création de la base de données

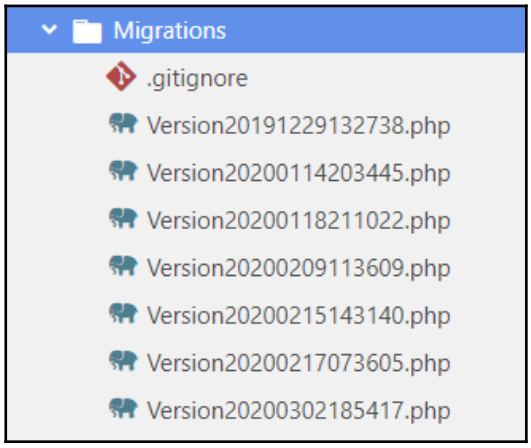

***Figure 10**: Dossier Migrations*

Le dossier Migrations (***Figure 10***) contient les différentes versions des migrations que nous avons utilisées pour modifier la base de données. Ces fichiers contiennent les instructions SQL pour réaliser la création de la base de données et les modifications que nous avons effectuées au long du développement. Ils sont dans notre cas générés automatiquement avec la commande 'bin/console make:migration'.

2.4.6. Modifier les requêtes à la base de données

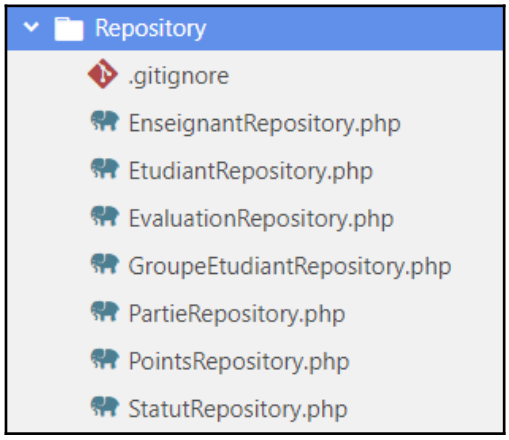

***Figure 11**: Dossier Repository*

Le dossier Repository (***Figure 11***) contient pour chaque entité des requêtes à la base de données portant sur une entité donnée. Ces fichiers contiennent des requêtes crées par défaut lors de la création d'une entité (findAll, findBy, find,…) ou des requêtes personnalisées dans certains cas. Dans le cas de l'entité GroupeEtudiant, on retrouve également des requêtes supplémentaires nécessaires pour gérer le comportement d'arbre que nous y avons appliqués, et que nous expliquerons dans la partie dédiée au code.

2.4.7. Modifier l'authentification à l'application

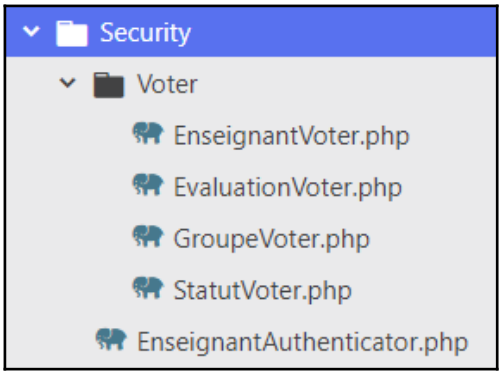

***Figure 12**: Dossier Security*

Le dossier Security (***Figure 12***) contient la classe permettant l'authentification des enseignants à leur profil. Cette classe a été générée automatiquement lors de la mise en place de l'authentification. Elle est adaptée à notre entité Enseignant, qui est une entité User, c'est-à-dire qui représente un profil dans l'application. On y retrouve aussi un dossier nommé Voter que nous évoquons dans la partie « Les Voters » de ce manuel.

2.5. Modifier l'interface de l'application

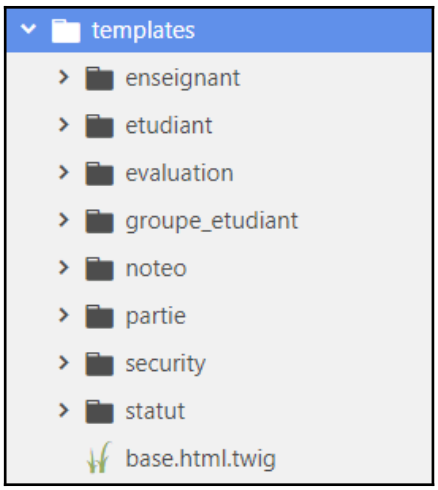

***Figure 13**: Dossier templates*

Le dossier templates (***Figure 13***) contient toute la partie vue de l'application. On y retrouve toutes les pages affichées, que nous avons regroupées par entité, par choix personnel. Toutes ces vues sont des extensions du fichier base.html.twig qui est un template contenant le squelette visuel de l'application. Nous utilisons la fonctionnalité des « blocks » de twig pour adapter certaines parties de ce template en fonction du contenu que nous voulons afficher.

3. Explication de parties complexes du code

3.1. Comportement d'arbre

Pour implanter la fonctionnalité permettant de créer des groupes, puis des sous-groupes, et des sous-groupes de ces sous-groupes et ainsi de suite, nous avons utilisé une partie du module StofDoctrineExtensionsBundle, plus particulièrement le module « tree ». Ce module nous permet d'appliquer un comportement d'arbre à l'entité GroupeEtudiant en intégrant des méthodes permettant de gérer des parents, enfants, nœuds, feuilles, et toutes les notions qui en découlent, pour pouvoir facilement gérer une telle décomposition. Nous avons réalisé l'installation du bundle avec composer, avec la commande suivante : 'composer require stof/doctrine-extensions-bundle'. Lien du module : https://symfony.com/doc/current/bundles/StofDoctrineExtensionsBundle/index.html

3.1.1. Explication du fonctionnement

Nous allons dans un premier temps expliquer la manière dont est représentée un arbre grâce à ce module (***Figure 14***). Nous prendrons un exemple simple : un groupe, DUT Info, et ses sous-groupes : S1, S2 et S3. On a également des sous-groupes pour S1 : TD1 et TD2. Cet arbre est donc représenté ainsi :

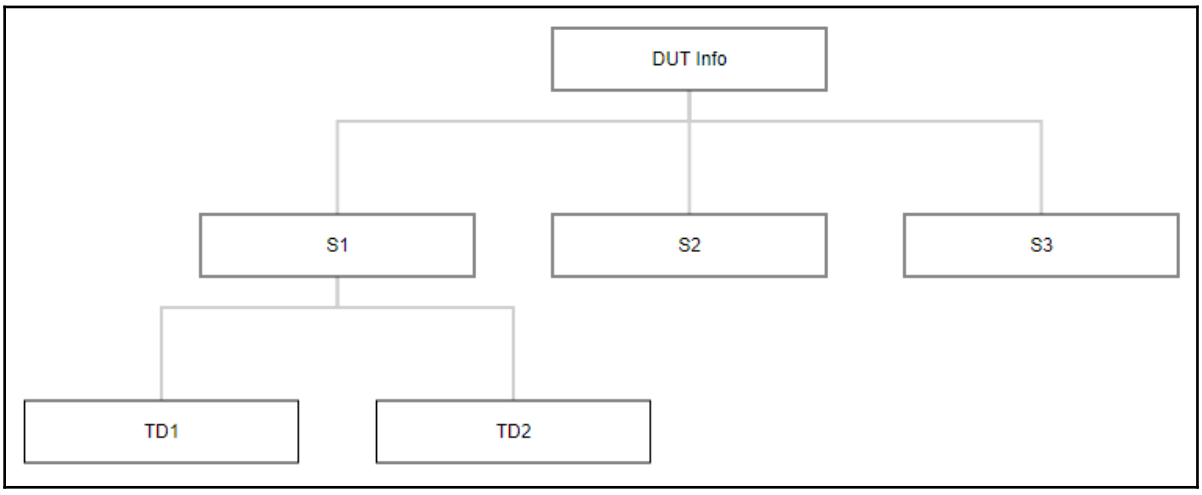

***Figure 14**: Arbre simple*

Si cet arbre était implémenté dans la base de donnée avec le module tree, sa représentation serait alors plutôt la suivante :

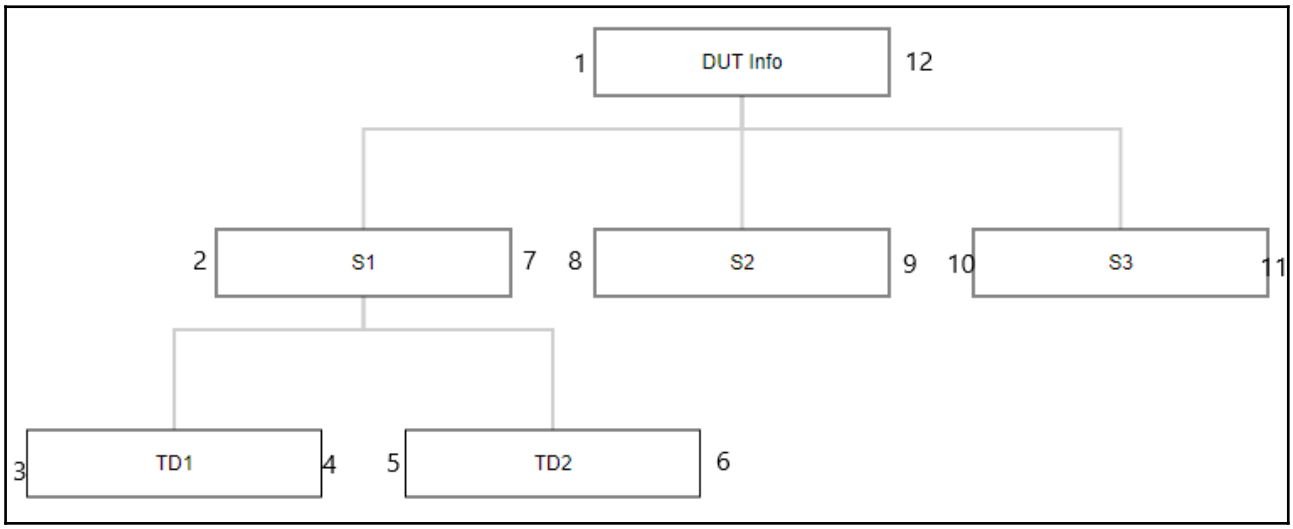

***Figure 15**: Application des fonctionnalités du module tree sur l'arbre de la figure 14.*

On remarque des nombres à gauche et à droite de chaque élément de l'arbre (***Figure 15***). Ces nombres, nommés explicitement « gauche » et « droite », permettent de déterminer l'ordre dans lequel chaque élément est placé, mais également quels éléments possèdent des sous-éléments et si oui lesquels. Le module gère également pour chaque élément son niveau (ici le niveau de DUT Info → 0, S1 → 1 et TD1 → 2 par exemple). Ce module permet donc de gérer des arbre avec des éléments imbriqués, et ordonnés.

3.1.2. Entité et repository

3.1.2.1. Entité

Au niveau de l'entité, du code, le comportement d'arbre est implanté via 6 attributs, avec leur accesseurs et mutateurs respectifs :

```php
/**
 * @Gedmo\TreeLeft
 * @ORM\Column(name="lft", type="integer")
 */
private $lft;

/**
 * @Gedmo\TreeLevel
 * @ORM\Column(name="lvl", type="integer")
 */
private $lvl;

/**
 * @Gedmo\TreeRight
 * @ORM\Column(name="rgt", type="integer")
 */
private $rgt;

/**
 * @Gedmo\TreeRoot
 * @ORM\ManyToOne(targetEntity="GroupeEtudiant")
 * @ORM\JoinColumn(name="tree_root", referencedColumnName="id", onDelete="CASCADE")
 */
private $root;

/**
 * @Gedmo\TreeParent
 * @ORM\ManyToOne(targetEntity="GroupeEtudiant", inversedBy="children", cascade={"persist"})
 * @ORM\JoinColumn(name="parent_id", referencedColumnName="id", onDelete="CASCADE")
 */
private $parent;

/**
 * @ORM\OneToMany(targetEntity="GroupeEtudiant", mappedBy="parent")
 * @ORM\OrderBy({"lft" = "ASC"})
 */
private $children;
```

***Listing 1**: Extrait de l'entité GroupeEtudiant (src/Entity/GroupeEtudiant.php)*

Les attributs que l'on peut lire dans le ***listing 1***, lft (left, gauche), rgt (right, droite) et lvl (level, niveau) aident, comme nous l'avons dit, le placement de chaque élément de l'arbre par rapport aux autres, plus particulièrement de connaître l'ordre dans lequel les éléments de l'arbre ont été créés, et leur niveau (la racine étant au niveau 0). Les attributs root, parent et children permettent respectivement de connaître pour chaque élément de n'importe quel arbre la racine de l'arbre, le parent et les enfants de l'élément.

3.1.2.2. Repository

Au niveau de l'entité GroupeEtudiant, l'utilisation des requêtes (récupération de parents, nœuds, enfants,...) propres au bundle se réalise en liant directement l'entité au repository intégrant les requêtes permettant de gérer l'arbre, nommé NestedTreeRepository. Il est créé automatiquement lors de l'installation du bundle et on le lie via une annotation spécifique (@ORM\Entity(repositoryClass=...)) comme on le voit sur le ***listing 2***.

```php
/**
 * @Gedmo\Tree(type="nested")
 * @ORM\Entity(repositoryClass="Gedmo\Tree\Entity\Repository\NestedTreeRepository")
 */
class GroupeEtudiant
{
```

***Listing 2**: En-tête de l'entité GroupeEtudiant (src/Entity/GroupeEtudiant.php)*

3.1.3. Modification et du listage des groupes étudiant, groupe des étudiants non affectés

Toutes les fonctionnalités expliquées ici et, sauf précision, les captures d'écran associées correspondent à des **extraits du contrôleur GroupeEtudiantController**. La création d'un groupe étudiant n'est pas abordée car elle peut être réalisée comme pour tout autre entité, les attributs « spéciaux » de l'arbre étant gérés automatiquement, il n'y a donc pas à les saisir nous même. La seule spécificité est l'ajout d'étudiants à partir d'un fichier CSV. De même pour la suppression d'un groupe étudiant, dont le seul point particulier est la nécessité de supprimer toutes les évaluations liées au groupe. La suppression se fait en cascade (comme on peut le voir avec l'attribut onDelete, dans la 3e annotation sur le ***listing 3***), il n'y a donc pas a se préoccuper une fois de plus des attributs spéciaux de l'arbre qui sont mis à jour automatiquement dans ce cas.

```php
/**
 * @Gedmo\TreeParent
 * @ORM\ManyToOne(targetEntity="GroupeEtudiant", inversedBy="children", cascade={"persist"})
 * @ORM\JoinColumn(name="parent_id", referencedColumnName="id", onDelete="CASCADE")
 */
private $parent;
```

***Listing 3**: Annotations de l'attribut « parent » (src/Entity/GroupeEtudiant.php)*

3.1.3.1. Lister les groupes étudiant

Nous avons décidé que, pour que l'utilisateur puisse s'y retrouver plus facilement dans les différents groupes de l'application créés par l'administrateur, nous afficherons les groupes et leurs enfants de manière hiérarchique, avec de l'indentation. Pour cela nous avons utilisé les propriétés du module tree avec une requête personnalisée (***Listing 4***).

```php
/**
 * @return GroupeEtudiant[] Returns an array of GroupeEtudiant objects
 */

public function findAllOrderedAndWithoutSpace()
{
    return $this->createQueryBuilder('g')
        ->addSelect('et')
        ->addSelect('en')
        ->join('g.enseignant', 'en')
        ->leftJoin('g.etudiants', 'et')
        ->where('g.slug != :param')
        ->setParameter('param', 'etudiants-non-affectes')
        ->orderBy('g.lft', 'asc')
        ->getQuery()
        ->getResult()
        ;
}
```

***Listing 4**: Récupération des groupes hiérarchiquement (src/Repository/GroupeEtudiantRepository.php)*

Nous ordonnons les résultats de la requête par l'attribut « lft » ce qui aura pour effet d'ordonner le résultat de la requête dans l'ordre hiérarchique des groupes. Par exemple : pour une arborescence contenant le groupe DUT Informatique, ses sous-groupes S1 et S2 et TD1, sous-groupe de S1, cette requête retourne, dans l'ordre : DUT informatique, S1, TD1, S2. Dans la vue, nous n'avons alors qu'a, en fonction du niveau, appliquer l'indentation que l'on désire en rajoutant des espaces à l'aide du symbole   ; (***Figure 16***).

```twig
{% if groupe.lvl > 0 %}
  {% for i in 1.. groupe.lvl %}
     
  {% endfor %}
{% endif %}
{{ groupe.nom }}
```

***Figure 16**: Affichage de l'indentation (templates/groupe/index.html.twig)*

3.1.3.2. Groupe des étudiants non affectés

A noter que dans la requête du **listing 4**, dans la fonction where(), on ne récupère pas le groupe dont le slug est égal à « etudiants-non-affectes ». Ce groupe est un groupe spécial, géré par l'application, qui contient les étudiants ne faisant pas partie de l'arborescence créée par l'administrateur. Sa création est gérée lors de la création du groupe de haut niveau (donc de l'arborescence des groupes de l'application) :

```php
//Si le groupe des étudiants non affectés n'existe pas déjà on le crée
if ($repo->findOneBySlug('etudiants-non-affectes') == null) {
    $nonAffectes = new GroupeEtudiant();
    $nonAffectes->setNom("Etudiants non affectés");
    $nonAffectes->setDescription("Tous les étudiants ayant été retirés d'un groupe de haut niveau et ne faisant partie d'aucun groupe");
    $nonAffectes->setEnseignant($this->getUser());
    $nonAffectes->setEstEvaluable(false);
    $entityManager->persist($nonAffectes);
}
```

***Listing 5**: Création du groupe des étudiants non affectés*

On ne le créera que si il n'existe pas déjà (***Listing 5***), car l'application est livrée sans groupe dans la base de donnée. Comme l'utilisateur n'a aucune action sur ce groupe, si tous les groupes de l'application sont supprimé, ce groupe subsistera. Si pour une raison ou une autre celui-ci est supprimé, il sera donc créé à nouveau lors de la création d'un nouveau groupe de haut niveau.

3.1.3.3. Modification des attributs et des étudiants d'un groupe étudiant

Le cas de la modification d'un groupe étudiant est particulier. En effet, sur la page de modification, en plus de pouvoir modifier les attributs d'un groupe, on remarque que l'on peut ajouter et supprimer des étudiants d'un groupe. Deux cas peuvent alors se produire en fonction du groupe modifié. Si il est de « haut niveau » (si c'est la racine d'un arbre), la liste d'ajout sera composée de la liste des étudiants non affectés à un groupe (qui, comme nous l'avons expliqué plus tôt sont dans un groupe invisible aux utilisateurs). On veut donc lors de la modification savoir si le groupe est un groupe de « haut niveau » ou non (pour cela on teste s'il a un parent : si non, il est de haut niveau, si oui, il ne l'est pas). Si c'est le cas, on pourra ajouter des étudiants depuis le groupe des étudiants non affectés, sinon depuis le parent du groupe sélectionné. Pour cela on passe le groupe en paramètre du formulaire de modification, et dans le formulaire on récupérera la liste de ses étudiants parmi lesquels l'utilisateur pourra choisir quels étudiants ajouter. On peut voir la manière dont est effectué ce choix dans le ***listing 6***.

```php
/* On prépare une variable qui contiendra le groupe à partir duquel ajouter les étudiants. En effet, si le groupe
est de haut niveau, on ajoute des étudiants depuis le groupe des étudiants non affectés, sinon on ajout des étudiants
depuis son parent (car dans ce cas, le groupe est un sous groupe) */
if ($groupeEtudiant->getParent() == null) {
  $groupeAPartirDuquelAjouterEtudiants = $GroupeDesNonAffectés;
}
else {
  $groupeAPartirDuquelAjouterEtudiants = $groupeEtudiant->getParent();
}
```

***Listing 6**: Définition du groupe à partir duquel ajouter des étudiants*

Pour modifier un groupe, on utilise un formulaire qui permettra de saisir le nom et la description, qui viennent directement hydrater l'entité GroupeEtudiant que l'on modifie. De plus, on peut sélectionner des étudiants parmi deux listes : une pour ajouter des étudiants (qui dépendra, comme expliqué, du groupe sélectionné), et une liste pour choisir quels étudiants supprimer du groupe, qui correspond à celle du groupe sélectionné.

```php
if ($groupeEtudiant->getParent() == null) {

  //Le groupe est de haut niveau alors on ajoute dans le groupe et on supprime du groupe des non affectés
  foreach ($form->get('etudiantsAAjouter')->getData() as $key => $etudiant) {
   $groupeEtudiant->addEtudiant($etudiant);
   $GroupeDesNonAffectés->removeEtudiant($etudiant);
  }

  //Le groupe est de haut niveau alors on supprime l'étudiant dans les sous-groupes et dans le groupe
  foreach ($form->get('etudiantsASupprimer')->getData() as $key => $etudiant) {
    foreach ($enfants as $enfant) {
      $enfant->removeEtudiant($etudiant);
    }
    $groupeEtudiant->removeEtudiant($etudiant);
    $GroupeDesNonAffectés->addEtudiant($etudiant);
  }

}
```

***Listing 7**: Ajout et suppression des étudiants dans le cas d'un groupe de haut niveau*

Une fois le formulaire validé, pour ajouter ou supprimer des étudiants on décompose une nouvelle fois l'action en deux cas : la modification d'un groupe de haut niveau et les autres. Dans le ***listing 7***, on peut voir le cas d'un groupe de haut niveau. Dans un premier temps on traite les étudiants sélectionnés dans la liste d'ajout (1er foreach). Ces étudiants sont ajoutés au groupe puis supprimés du groupe des non affectés. Puis on traite ceux sélectionnés dans la liste de suppression (2e foreach). Ces étudiants sont supprimés de tous les enfants du groupe modifié (par enfant on entend **tous** les groupes de l'application car notre application ne comporte qu'un arbre de groupe et on supprime des étudiants de la racine) puis ajoutés dans le groupe des non affectés.

```php
else {

  //Le groupe n'est pas de haut niveau alors on ajoute juste l'étudiant dans le sous-groupe
  foreach ($form->get('etudiantsAAjouter')->getData() as $key => $etudiant) {
   $groupeEtudiant->addEtudiant($etudiant);
  }

  //Le groupe n'est pas de haut niveau alors on supprime juste l'étudiant dans le sous-groupe et ses sous-groupes
  foreach ($form->get('etudiantsASupprimer')->getData() as $key => $etudiant) {
    //On supprime l'étudiant des sous groupes
    foreach ($enfants as $enfant) {
      $enfant->removeEtudiant($etudiant);
    }
    $groupeEtudiant->removeEtudiant($etudiant);
  }

}
```

***Listing 8**: Ajout et suppression des étudiants dans le cas d'un groupe qui n'est pas de haut niveau*

Dans le ***listing 8***, on traite l'ajout et la suppression des étudiants dans le cas d'un groupe qui n'est pas de haut niveau. Pour chacun des étudiants que l'on a sélectionnés dans la liste d'ajout (composée de la liste du parent du groupe, dans ce cas) on l'ajoute simplement dans le groupe modifié. De même que pour le haut niveau on supprime l'étudiant de tous les enfants du groupe modifié et du groupe modifié. Mais on ne le supprime pas du parent, selon les règles de gestion que nous avons choisies.

3.2. Création d'une évaluation et consultation des statistiques d'une évaluation

Toutes les fonctionnalités expliquées ici et, sauf précision, les captures d'écran associées correspondent à des **extraits du contrôleur EvaluationController**. De même que pour les groupes, nous n'aborderons pas la suppression qui n'est pas particulièrement complexe si ce n'est que nous devons penser à supprimer pour l'évaluation chaque partie et les notes associées (comme on peut le déduire selon la structure de notre diagramme de classe sur la ***figure 17***). La récupération et l'affichage de toutes les évaluations se fait également « classiquement ». La modification n'est pas non plus abordée car celle-ci est similaire à la création.

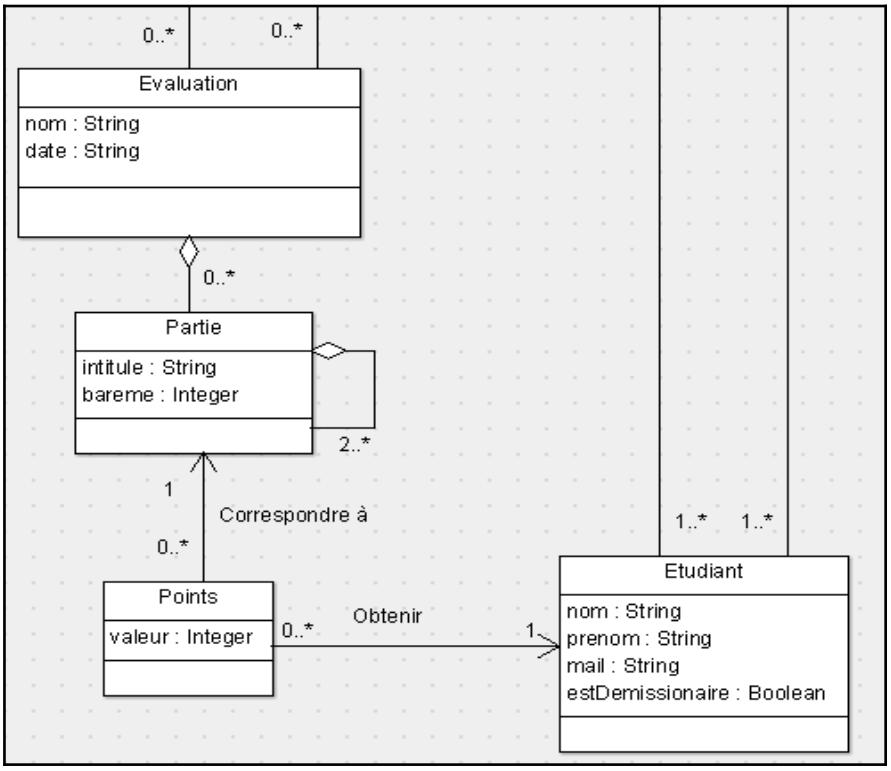

***Figure 17**: Extrait du diagramme de classes*

3.2.1. Création d'une évaluation

La création d'une évaluation est un cas particulier. En effet, pour la création d'une évaluation nous avons besoin de pouvoir saisir les notes de celle-ci même si comme nous avons pu le voir dans l'extrait du diagramme de classes de la ***figure 17***, les liens entres les entités existent, et nous permettent donc théoriquement de récupérer les informations dont nous avons besoin pour pouvoir saisir les notes d'une évaluation donnée, en pratique le fait d'avoir une entité à part « Points » correspondant à un étudiant et une partie rend compliquée la saisie des attributs d'une évaluation et de la valeur du « point » dans un même formulaire portant sur l'évaluation. Nous avons donc pour résoudre ce problème adopté la solution visible dans le ***listing 9***.

```php
public function new(Request $request, GroupeEtudiant $groupeConcerne, ValidatorInterface $validator): Response
{
    //Création d'une évaluation vide avec tous ses composants (partie, notes(définies à 0 par défaut))
    $evaluation = new Evaluation();
    $evaluation->setGroupe($groupeConcerne);
    $partie = new Partie();
    $partie->setIntitule("");
    $partie->setBareme(20);
    $evaluation->addPartie($partie);
    foreach ($groupeConcerne->getEtudiants() as $etudiant) {
      $note = new Points();
      $note->setValeur(0);
      $etudiant->addPoint($note);
      $partie->addNote($note);
    }

    //Création du formulaire pour saisir les informations de l'évaluation (le formulaire n'est pas lié à une en
    $form = $this->createFormBuilder(['notes' => $partie->getNotes()])
        ->add('nom', TextType::class)
        ->add('date', DateType::class, [
          'widget' => 'single_text'
        ])
        ->add('notes', CollectionType::class , [
          'entry_type' => PointsType::class //Utilisation d'une collection de formulaire pour saisir les valeur
                                            //passées en paramètre du formulaire)
        ])
        ->getForm();
```

***Listing 9**: Extrait de la création du formulaire de création d'une évaluation*

Dans un premier temps, comme on peut le voir dans les premières lignes du ***listing 9***, on crée une entité Evaluation « vide » qui sera hydratée par le formulaire. On définit les informations que l'on connaît déjà comme le groupe concerné, et la partie associée (comme il s'agit ici d'une évaluation simple, on crée une évaluation avec une unique partie, et celle-ci ne sera pas modifiable ni visible par l'enseignant). On crée également, pour chaque étudiant du groupe sur lequel porte l'évaluation, une entité Points, que l'on lie à la partie et à l'étudiant et dont on définit la note à 0, en valeur « par défaut ».

Dans un deuxième temps, on crée un formulaire, non lié à une entité, qui permettra de récupérer les informations pour hydrater l'entité Evaluation (nom et date). Pour saisir les notes, on utilise un champ de formulaire de type Collection, qui sera une collection de formulaires portant sur l'entité Points. Ce qui veut dire que pour chaque entité Points, un formulaire permettant de saisir sa valeur sera affiché, et hydratera les entités Points (on a donc des formulaires portant sur des entités Points imbriqués dans le formulaire créé dans le ***listing 9***). Pour que le formulaire sache combien de formulaires créer et sur quelles entités Points les faire porter, on passe en paramètre du formulaire un tableau d'entité points : $this->createFormBuilder(['notes' => $partie->getNotes()]). Cette ligne veut dire que le champ du formulaire notes (la collection de formulaires), se basera sur toutes les entités Points qui constituent les notes des étudiants précédemment crées à la partie de l'évaluation. Pour chacune de ces entités un formulaire pour saisir sa note sera donc créé.

```php
if ($form->isSubmitted()) {

    $entityManager = $this->getDoctrine()->getManager();

    $data = $form->getData(); //Récupération des données du formulaire

    $evaluation->setNom($data["nom"]); // Définition du nom de l'évaluation
    $evaluation->setDate($data["date"]); // -------- de la date -----------
    $evaluation->setEnseignant($this->getUser());

    //Validation de l'entité hydratée à partir des données du formulaire
    $this->validerEntite($evaluation, $validator);
    $this->validerEntite($partie, $validator);

    $entityManager->persist($evaluation);
    $entityManager->persist($partie);

    foreach ($partie->getNotes() as $note) {

      //Si la note dépasse le barême de la partie, on réduit la note à la valeur du barême
      if ($note->getValeur() > $partie->getBareme()) {
        $note->setValeur($partie->getBareme());
      }
      if ($note->getValeur() < 0) {
          $note->setValeur(0);
      }
      //On valide l'entité note hydratée avec la collection de formulaires
      $this->validerEntite($note, $validator);
      $entityManager->persist($note);
    }

    $entityManager->flush();
    return $this->redirectToRoute('evaluation_enseignant',['id' => $this->getUser()->getId()]);
}

return $this->render('evaluation/new.html.twig', [
    'evaluation' => $evaluation,
    'form' => $form->createView(),
]);
```

***Listing 10**: Enregistrement et Validation des informations soumises du formulaire*

Une fois le formulaire est validé, on peut maintenant hydrater l'entité Evaluation avec le nom et la date saisis (***Listing 10***). On valide manuellement les entités Partie, Points et Evaluation avec une méthode que nous avons créée (validerEntite()) qui permet de faire appel à la fonction de l'entité Validator pour valider les entités selon les règles définies dans le fichier des entités correspondantes, comme par exemple celles que l'on peut voir dans le ***listing 11*** (de la form @Assert\....). On vérifie également pour chaque note si celle-ci est supérieure au barème défini, et si c'est le cas on la définit comme égale au barème. Si elle est inférieure à 0, on la définit comme égale à 0. Cette vérification est là comme une « sécurité » supplémentaire avant la validation de l'entité.

```php
/**
 * @ORM\Column(type="string", length=50)
 * @Assert\Length(max=50)
 * @Assert\NotBlank
 */
private $nom;

/**
 * @ORM\Column(type="date")
 * @Assert\NotBlank
 * @Assert\Date
 */
private $date;
```

***Listing 11**: Extrait de l'entité Evaluation (src/Entity/Evaluation.php)*

3.2.2. Statistiques

Les deux listings présentés dans cette partie se situent dans le fichier **src/Controller/EvaluationController.php**. L'affichage des statistiques se réalise en deux temps : on choisit d'abord les différents groupes et statuts sur lesquels on veut visionner des statistiques, puis les statistiques sont affichées, en fonction des groupes et statuts choisis. Bien que ce ne soit pas le cas, dans la première étape les statuts sont considérés comme des groupes au niveau du code pour plus de simplicité, mais ceux-ci sont bien référencés comme statuts dans l'interface utilisateur.

```php
public function chooseGroups(Request $request, $idEval, $idGroupe, StatutRepository $repoStatut, EvaluationRepository $rep
{
    //On récupere l'évaluation que l'on traite pour afficher ses informations générales dans les statistiques
    $evaluation = $repoEval->find($idEval);

    //On récupère le groupe concerné par l'évaluation
    $groupeConcerne = $repoGroupe->find($idGroupe);

    //On ajoute dans un tableau le groupe concerné ainsi que tous ses enfants, pour pouvoir choisir ceux sur lesquels ont
    $choixGroupe[] = $groupeConcerne;
    foreach ($this->getDoctrine()->getRepository(GroupeEtudiant::class)->children($groupeConcerne, false) as $enfant) {
      $choixGroupe[] = $enfant;
    }

    //Création du formulaire pour choisir les groupes / status sur lesquels on veut des statistiques
    $form = $this->createFormBuilder()
        ->add('groupes', EntityType::class, [
          'class' => GroupeEtudiant::Class, //On veut choisir des groupes
          'choice_label' => false, // On n'affichera pas d'attribut de l'entité à côté du bouton pour aider au choix car c
          'label' => false, // On n'affiche pas le label du champ
          'mapped' => false, // Pour que l'attribut ne soit pas immédiatement mis en BD mais soit récupérable après soumis
          'expanded' => true, // Pour avoir des cases
          'multiple' => true, // à cocher
          'choices' => $choixGroupe // On choisira parmis le groupe concerné et ses enfants
        ])
        ->add('statuts', EntityType::class, [
          'class' => Statut::Class,
          'choice_label' => false,
          'label' => false,
          'mapped' => false,
          'expanded' => true,
          'multiple' => true,
          'choices' => $repoStatut->findAll() // On choisira parmis tous les statut
        ])
        ->getForm();

    $form->handleRequest($request);
```

***Listing 12**: Création du formulaire de choix des groupes et statuts*

Pour la première étape, le choix des groupes et statuts, on commence donc par récupérer l'évaluation correspondante, ainsi que le groupe associé à l'évaluation, comme on le voit dans les premières lignes du ***listing 12***. Puis on crée un formulaire avec deux champs EntityType (pour pouvoir, une fois le formulaire soumis, récupérer directement les entités choisies). Le premier champ correspond à la liste où l'on pourra sélectionner les groupes, la liste étant composée du groupe sur lequel porte l'évaluation et ses enfants. Le deuxième champ correspond à la liste où l'on pourra sélectionner les statuts, parmi tous ceux existant.

```php
if ($form->isSubmitted()) {

    $listeStatsParGroupe = array(); // On initialise un tableau vide qui contiendra les

    $listeStatsParStatut = array(); // On initialise un tableau vide qui contiendra les

    //Pour tous les groupes sélectionnés
    foreach ($form->get("groupes")->getData() as $groupe) {

        //On récupère toutes les notes du groupe courant
        $tabPoints = $repoPoints->findByGroupe($idEval, $groupe->getId());

        //On crée une copie de tabPoints qui contiendra les valeurs des notes pour simpl
        $copieTabPoints = array();
        foreach ($tabPoints as $element) {
            $copieTabPoints[] = $element["valeur"];
        }

        //On remplit le tableau qui contiendra toutes les statistiques du groupe
        $listeStatsParGroupe[] = array("nom" => $groupe->getNom(),
                                       "notes" => $this->repartition($copieTabPoints),
                                       "moyenne" => $this->moyenne($copieTabPoints),
                                       "ecartType" => $this->ecartType($copieTabPoints),
                                       "minimum" => $this->minimum($copieTabPoints),
                                       "maximum" => $this->maximum($copieTabPoints),
                                       "mediane" => $this->mediane($copieTabPoints)
                                       );
    }
```

Listing 13*: Création des statistiques pour tous les groupes choisis*

Comme on le voit dans le ***listing 13***, une fois le formulaire validé, on prépare deux tableaux. Ces tableaux contiendront les statistiques des groupes et des statuts choisis. Pour pour chaque groupe et statut choisi, on récupérera toutes les notes obtenues l'évaluation par les étudiants faisant partie du groupe ou du statut (on crée également une copie de ce tableau plus adapté pour le traitement par la suite car la structure de tableau retournée ne nous convenait pas). Ensuite, pour chaque groupe/statut, on ajoute dans le tableau des statistiques correspondant un tableau contenant les statistiques (avec en plus le nom) du groupe ou du statut. Les tableaux de statistiques des groupes et des statuts sont ensuite fusionnés dans un seul tableau qui nous servira pour afficher les statistiques par groupe dans la page visible par l'utilisateur.

3.3. Datatables

Pour pouvoir bénéficier de fonctions de tri, pagination et recherche dans un tableau html, nous avons décidé d'utiliser une bibliothèque nommée Datatables (https://datatables.net/). Déjà intégrée dans le template Bootstrap que nous avons choisi, Datatables est paramétrable à l'aide d'une fonction JavaScript et d'options au format JSON. Cette fonction s'applique directement à un tableau html.

Un exemple de paramétrage dans notre code est le suivant (#table est l'identifiant du tableau sur lequel on active les fonctionnalités) :

```
<script type="text/javascript">
  $(document).ready(function() {
    $('#table').dataTable({
      language: {
        // Suppression du label Rechercher et ajout du placeholder
        search: "_INPUT_",
        searchPlaceholder: "Rechercher...",
        "url": "{{asset('tradfr.json')}}"
      },
      columns: [null, null, null, null, {"orderable": false}],
      order: [
        [0, "asc"]
      ],
      lengthMenu: [
        [10, 25, 50, -1],
        [10, 25, 50, "Tout"]
      ],
      info: false,
      columnDefs: [{
        "sType": "date-eu",
        targets: 1
      }]
    });
  });
</script>
```

***Listing 14**: Fonction d'initialisation des fonctions de Datatables sur un tableau (templates/evaluation/index.html.twig)*

Les options (dont l'on voit un exemple ***listing 14***) que nous utilisons sont les suivantes :

- **language** : modification de la langue des mots affichés. L'option « url » contient un lien vers un fichier de traduction traduisant tous les termes, et nous en avons rajouté un de plus pour des raisons pratiques (searchPlaceholder).
- **columns** : Cette option définit quelle colonne sera triable ou non (null étant oui car non renseigné, { "orderable": false } étant non). Si cette option est utilisée, on doit retrouver autant de valeurs que de colonnes dans le tableau.

- **order** : définit la colonne qui sera triée par défaut lors de l'affichage, dans l'ordre ascendant. Le numéro à renseigner est celui de la colonne -1. La première colonne sera donc la colonne 0.
- **lengthMenu** : définit dans la liste déroulante en haut à gauche les différentes tailles de page disponibles. Le premier tableau correspond aux valeurs réelles (-1 correspond à tout), le deuxième correspond à ce qui sera affiché
- **info** : désactive certaines informations sur la page courante. De la même manière il est possible de désactiver la pagination (paging), le tri sur toutes les colonnes (ordering) et la recherche (searching)
- **columnDefs** : Cette option spéciale est utilisée seulement dans le listage des évaluations. Il s'agit d'une extension que nous avons utilisée pour pouvoir trier les évaluations par date car le tri par ordre sémantique de Datatables ne convenait pas.

Un point important à soulever est que nous avons, lors du développement rencontré un problème avec une des fonctionnalités de Datatables. En effet, pour réaliser sa pagination, Datatables retire des lignes du document DOM. Ce qui veut dire que si l'on se trouve dans la page 2, seules les lignes de la page 2 seront présentes dans le code source de la page. Ceci a posé problème pour la soumission de certain formulaires pour lesquels nous voulions récupérer toutes les lignes du tableau. Nous sommes passé outre ce problème à l'aide de la fonction JavaScript suivante (***Figure 18***) :

```
$("[name='form']").on('submit', function () {
  table.rows().nodes().page.len(-1).draw();
});
```

Figure 18*: Affichage de toutes les lignes d'un tableau (templates/evaluation/new.html.twig)*

Pour un formulaire de nom « form » cette fonction, lors de la soumission, affiche toutes les lignes du tableau dans une même page, ce qui permet d'avoir tous les résultats entrés dans le tableau dans le code source de la page et donc de récupérer tous les résultats.

3.4. Chart.js

Dans le fichier templates/evaluation/stats.html.twig, pour pouvoir afficher les statistiques, nous avons opté pour l'utilisation de la bibliothèque Chart.js (https://www.chartjs.org/) qui nous permet de bénéficier d'une génération simple de graphiques en barres (pour la répartition des notes), avec comme Datatables un paramétrage à l'aide d'une fonction JavaScript et de paramètres au format JSON. Cette fonction est appliquée à une balise canvas (***Figure 19***) pour préciser où le graphique sera affiché.

```
<canvas id="{{groupe.nom}}"></canvas>
```

Figure 19*: Création d'un canvas*

```
var graph = document.getElementById('{{groupe.nom}}').getContext('2d');

var leBarre = new Chart(graph, {
  type: 'bar',
  data: {
    labels: ['[0;4[', '[4;8[', '[8;12[', '[12;16[', '[16;20]'],
    datasets: [{
      label: 'Effectif',
      data: {{groupe.notes|json_encode()}}, {# json_encode sert à rendre le tableau lisible en JavaScript #}
      backgroundColor: "#4e73df",
      hoverBackgroundColor: "#2e59d9",
      borderColor: "#4e73df"
    }]
  },
```

***Listing 15**: Affichage du graphique*

Comme on le voit sur le ***listing 15*** on n'applique pas directement la fonction au canvas mais on récupère d'abord ce canvas dans une variable. On le récupère via son id (qui est égal au nom du groupe/statut, ce qui aide pour la création des graphiques car cette fonction sera écrite autant de fois qu'il y a de groupes/status). Ensuite, crée une nouvelle instance de la classe Charte en passant au constructeur en premier paramètre le canvas récupéré précédemment, puis en deuxième paramètre les options au format JSON. Les options les plus importantes sont les suivantes :

- **type** : définit le type de graphique qui sera affiché, dans notre cas un histogramme (bar).
- **data** : définit le jeu de données que le graphique affichera, et les options d'affichage. Dans notre cas, pour chaque valeur x de « labels », nous devons fournir une valeur y correspondante dans « data ». Pour cela nous avons déjà calculé 5 valeurs dans EvaluationController qui correspondent au nombre de notes comprises dans chaque intervalle. Ces valeurs sont passées dans un tableau à la vue, comme les autres statistiques, comme nous l'avons expliqué dans la partie dédiée aux évaluations. Nous utilisons la fonction json_encode pour transformer ce tableau au format json, qui sera lisible par la fonction javascript. En effet, toutes les options du graphique sont au format json.
- Les autres options sont des options graphiques qui permettent de définir l'épaisseur des barres, la transparence, les couleurs, les légendes... Pour les paramétrer efficacement nous recommandons une lecture de la documentation de la bibliothèque à l'adresse suivante : https://www.chartjs.org/docs/latest/configuration.

3.5. Les voters

Pour certaines des actions métiers que nous avons mises en places dans les controleurs, nous avions besoin d'un contrôle d'accès particulier. Par exemple, il nous était impossible dans le fichier config/packages/security.yaml de restreindre l'accès à toutes les fonctionnalités des profils de l'application seulement à l'administrateur. En effet, même si un enseignant ne peut créer/supprimer/modifier/consulter TOUS les profils, il peut modifier ou consulter le sien. Nous avons donc, pour ce type de cas (que l'on peut retrouver par exemple dans le fait de pouvoir modifier les évaluations que l'on a créé mais pas celles des autres), mis en place un mécanisme de symfony nommé Voter. Un voter est une classe, écrite dans un fichier php, sous le formalisme suivant : EntitéVoter.php. Ces fichiers sont stockés dans src/Security/Voter.

Un voter est appelé dans un controleur via la méthode $this-> denyAccessUnlessGranted(). Cette fonction prend deux paramètre : l'action réalisée (sous forme de chaine de caractères) et une instance de l'entité sur laquelle elle est réalisée (par exemple Evaluation). Par exemple, dans le controleur d'évaluation sur la méthode de modification d'une évaluation (***Listing 16***).

```php
/**
 * @Route("/modifier/{slug}", name="evaluation_edit", methods={"GET","POST"})
 */
public function edit(Request $request, Evaluation $evaluation, ValidatorInterface $validator): Response
{
  $this->denyAccessUnlessGranted('EVALUATION_EDIT', $evaluation);
```

***Listing 16**: Appel d'un voter*

Cette fonction aura pour effet d'appeler tous les voters créés dans l'application. Comme nous en avons plusieurs (par exemple GroupeEtudiantVoter et EvaluationVoter), il faut pouvoir savoir quelle voter doit choisir, ou non, de donner l'accès à cette page en fonction des conditions. Ainsi, on a dans le voter la méthode supports(). Cette méthode récupère les deux paramètres de la fonction denyAccessUnlessGranted(). Son but est de déterminer en fonction de sa valeur de retour (true/false), si le voter peut autoriser ou non l'accès à l'action métier du controleur. Pour cela elle détermine si l'action indiquée au voter est valide (si elle fait bien partie d'une liste prédéfinie) et si l'entité sur laquelle elle est réalisée est une entité correcte (c'est à dire si c'est bien une instance de l'entité de notre choix). Un exemple pour le voter de l'entité Evaluation (src/Security/EvaluationVoter.php) est lisible dans le ***listing 17***.

```php
protected function supports($attribute, $subject)
{
    return in_array($attribute, ['EVALUATION_PREVISUALISATION_MAIL', 'EVALUATION_EXEMPLE_MAIL', 'EVALUATION_ENVOI_MAIL', 'EVALUATION_EDIT', 'EVALUATION_DELETE'])
        && $subject instanceof \App\Entity\Evaluation;
}
```

***Listing 17**: Fonction supports()*

Si cette fonction renvoie true, alors la fonction voteOnAttribute() du voter est appelée. En fonction de la valeur de retour de cette fonction (true/false), l'accès sera autorisé, ou une erreur AccessDenied sera jetée et l'utilisateur n'aura pas accès à la fonctionnalité. Toujours pour le voter de l'entité Evaluation, un exemple de la fonction VoteOnAttribute() est lisible dans le ***listing 18***.

```php
protected function voteOnAttribute($attribute, $subject, TokenInterface $token)
{
    $user = $token->getUser();

    if (!$user instanceof UserInterface) {
        return false;
    }

    $accesAutorise = true;
    switch ($attribute) {
        case 'EVALUATION_PREVISUALISATION_MAIL':
        case 'EVALUATION_EXEMPLE_MAIL':
        case 'EVALUATION_ENVOI_MAIL':
        case 'EVALUATION_EDIT':
        case 'EVALUATION_DELETE':
            //Pour les 5 cas, on vérifie on a accès a la fonctionnalité si on est admin ou propriétaire de l'évaluation
            $accesAutorise = $accesAutorise = in_array("ROLE_ADMIN", $user->getRoles()) || $subject->getEnseignant()->getId() == $user->getId();
            break;
    }

    return $accesAutorise;
}
```

***Listing 18**: Fonction voteOnAttribute()*

La fonction teste tout d'abord si l'utilisateur est bien authentifié (sinon l'accès n'est pas autorisé). Ensuite, via un switch contenant toutes les actions définies dans la liste lisible dans la fonction supports(), on précise un à un la logique pour accorder ou non l'accès. Dans le cas de l'évaluation, on ne peut ajouter/modifier/consulter/supprimer une évaluation que si l'on est admin (un admin a contrôle sur toutes les évaluation) ou qu'il s'agit de l'évaluation créée par l'utilisateur connecté. De même pour la prévisualisation et l'envoi du mail aux étudiants. Via ces 5 cas nous avons restreint l'accès aux 5 méthodes métiers accessibles par l'utilisateur via les routes. Cependant la méthode denyAccessUnlessGranted doit être appelée avec les bon paramètres (indiquant l'action réalisée) dans chaque méthode métier pour que le contrôle soit déclenché dans la méthode.