Développement d'une application de gestion de cours avec AngularJS

Keran Kocher

30 mars 2015

Table des matières

1 Introduction **3**

2 Démarrage **5**

3 AngularJS **7**
3.1 Introduction . . . 7
3.2 Spécificités et avantages . . . 8

4 Fonctionnalités **13**
4.1 Les professeurs . . . 13
4.2 Les étudiants . . . 15

5 Schémas **19**

6 Modèle relationnel **23**
6.1 Introduction . . . 23
6.2 Les cours . . . 24
6.3 Les chapitres . . . 26
6.4 Les commentaires . . . 27
6.5 La progression . . . 27

7 RestLess **29**
7.1 Présentation . . . 29
7.2 Les vues génériques . . . 29
7.3 Fonctionnement de RestLess . . . 31

8 MathJax **35**
8.1 Installation . . . 35
8.2 Personnalisation . . . 36

9 Tests **39**

10 Guide du développeur **41**
10.1 Fichiers . . . 41
10.2 URL . . . 42
10.3 Concepts . . . 43

11 Bogues et améliorations **45**
11.1 Bogues . . . 45
11.2 Améliorations . . . 45

12 Conclusion **47**

13 Bibliographie **49**
13.1 Bibliographie 49
13.2 Webographie 49

14 Source des illustrations **51**

CHAPITRE 1

Introduction

Bla

CHAPITRE 2

Démarrage

Prérequis

— Python 3 et pip installé

Installer Django et les dépendances

```
pip3 install -r requirements.txt
```

De plus, il faut encore installer ces paquets sur la machine :

— pandoc
— pdflatex

Lancer les migrations

```
python3 manage.py migrate
```

Créer un super utilisateur

Si besoin de se connecter à la zone d'administration ("/admin")

`python3 manage.py createsuperuser` et suivre les instructions

Lancer le serveur Django

```
python3 manage.py runserver
```

Données

— Créer les données nécessaires : `python3 manage.py seed`
— Finalement vous pouvez vous rendre sur l'URL `/courses` et profiter des fonctionnalités.

Tests

Pour lancer les tests, installer d'abord Protractor (ne pas lancer le serveur webdriver) : http://angular.github.io/protractor/#/tutorial

Ensuite lancer la commande `python3 manage.py tests`

CHAPITRE 3

AngularJS

3.1 Introduction

AngularJS est un framework JavaScript créé et maintenu depuis 2009 par Google, l'entreprise à l'origine du populaire moteur de recherche, du client Gmail mais aussi de beaucoup d'outils pour les développeurs. Outre le fait de créer des outils de programmation comme AngularJS, Google propose des outils d'analyse (Google Analytics [1]), de stockage (Google Cloud [2]), ou encore des serveurs pour héberger des applications. Une entreprise très présente non seulement dans le domaine public, mais également dans le soutien et la recherche des technologies informatiques. Pour revenir au sujet qui nous intéresse, l'équipe de Google a donc développé le framework AngularJS. Un framework est un ensemble de structures et d'outils programmés dans un langage et réutilisables, qui permettent de faciliter la construction d'applications. On trouve par exemple chez PHP le framework Symfony [3], chez Ruby Ruby on Rails [4], chez Python *Django* que nous utilisons pour notre site web. Finalement, chez JavaScript on trouve *AngularJS*, mais encore comme concurrent Ember [5] ou React [6].

AngularJS permet de développer une application web complète. C'est-à-dire qu'il offre plusieurs outils indispensables. Tout d'abord, un système de routes qui permet de lier des URL avec des pages différentes. En clair, l'on peut dire à notre application que lorsque l'utilisateur entre `monsite.com/contact` dans son navigateur, le fichier `contact.html` doit être affiché. Il y a aussi la prise en charge des formulaires, avec la possibilité de récupérer les informations entrées par l'utilisateur ou de faire une validation du formulaire, c'est-à-dire de vérifier les informations entrées. En revanche, AngularJS ne sait pas comment communiquer directement avec une base de données - une base de données est l'endroit où sont stockées les données persistantes -, il est seulement capable d'utiliser les données d'une base de données fournie par un serveur intermédiaire. Les bases de données sont souvent indispensables, car elles permettent de concevoir des sites web dynamiques. Il est donc nécessaire d'utiliser un autre langage avec AngularJS. Django jouera ce rôle mais nous éclaircirons ce concept plus tard lorsque nous étudierons son intégration avec Angular.

Pour bien comprendre la documentation du projet, il faut savoir ce qu'est Django. Comme mentionné précédemment, il s'agit d'un framework Python. Il est spécialisé dans la création de site web et est le leader face à ses concurrents comme Turbo Gears [7]. Django est ce que nous utilisons principalement pour créer notre application web. Créer un site web avec Python seulement est difficile et peu pratique, un framework est quasiment indispensable. Effectivement, Django fournit les outils nécessaires pour créer des pages HTML, communiquer avec une base de données, gérer les URLs, etc. Pour comprendre les exemples Django qui apparaitront, il faut savoir que celui-ci utilise le modèle MVC - modèle, vue, contrôleur - qui est expliqué dans la section suivante. La particularité est que Django a changé les termes : *modèle* reste *modèle*, *vue* est égale à *template* et *contrôleur* est égal à *vue*.

1. http://google.com/analytics. Consulté le 25 décembre 14.
2. https://cloud.google.com. Consulté le 25 décembre 14.
3. http://symfony.com. Consulté le 25 décembre 14.
4. http://rubyonrails.com. Consulté le 25 décembre 14.
5. http://emberjs.com. Consulté le 25 décembre 14.
6. http://facebook.github.io/react. Consulté le 10 mars 15.
7. http://www.turbogears.org. Consulté le 25 décembre 14.

MVC	MVT (Django)
Modèle	Modèle
Vue	Template
Contrôleur	Vue

3.2 Spécificités et avantages

Qu'est-ce qui démarque AngularJS de ses concurrents ? En bref, quelles sont ses fonctionnalités, qui facilitent tant la vie des développeurs ? La première chose à connaitre d'AngularJS est qu'il encourage le modèle MVC - modèle, vue, contrôleur. Il s'agit en fait de séparer dans notre application ces trois composants. Grossièrement, le modèle est la partie qui gère les données de l'application, la vue s'occupe de ce qui est affiché à l'utilisateur et le contrôleur est la partie qui relie le modèle à la vue. Il va chercher les données dans les modèles pour les donner à la vue qui les affiche et, vice-versa, il reçoit les données de la vue, par un formulaire par exemple, pour mettre à jour le modèle. Un principe très important dans ce motif de programmation est que le minimum de code logique doit se trouver dans les vues. Elles doivent se contenter d'afficher, car toutes les opérations se font idéalement dans les contrôleurs ou les modèles. L'avantage de cette organisation est une compréhension plus aisée et une modularité du code.

Dans la suite de cette section nous allons nous intéresser à quelques fonctionnalités clés d'AngularJS.

3.2.1 Du HTML expressif

Abordons d'abord les vues. Le premier "miracle" du framework est de transformer le HTML statique en un langage expressif et dynamique. A la base, le HTML est une syntaxe qui permet de structurer une page web à l'aide de balises qui définissent leur contenu, comme par exemple les balises qui délimitent les paragraphes, les titres, les images, etc. Traditionnellement, on utilise un autre langage, comme Python, pour transformer la page et y insérer le contenu dynamique qui provient de la base de données. Concrètement, il pourrait s'agir d'afficher une liste d'articles dans une page. Si l'on travaille avec Django, on assigne à une variable `articles` tous les articles avec une requête SQL ; SQL est le langage pour communiquer avec une base de données relationnelle. On fait ensuite une boucle dans la page HTML et l'on affiche le titre ainsi que le contenu de chaque article.

```
<html>
    <head></head>
    <body>
        <div id="articles">
            {% for article in articles %}
                <h1>{{ article.titre }}</h1>
                <p>{{ article.contenu }}</p>
            {% endfor %}
        </div>
    <body>
<html>
```

Les balises `{% %}` et `{{ }}` signifient simplement que ce n'est plus du HTML, mais que du code Django est à l'intérieur et doit être exécuté par le serveur.

Avec AngularJS, le modèle est légèrement différent. En effet, on trouve dans le framework ce que l'on appelle des directives, un concept spécifique à Angular. Ce sont des balises ou des attributs HTML supplémentaires que fournit AngularJS. On peut les ajouter à notre page et elles effectuent différentes actions selon leur rôle. Pour reprendre l'exemple des articles, pour faire une boucle, on utilise la directive `ng-repeat`. On l'utilise comme un simple attribut HTML.

```
<html>
    <head></head>
    <body>
        <div id="articles" ng-repeat="article in articles">
            <h1>{{ article.titre }}</h1>
            <p>{{ article.contenu }}</p>
        </div>
```

```
    <body>
<html>
```

A l'instar de Django, les doubles accolades signifient que l'on veut exécuter du code Angular/JavaScript à l'intérieur. Ici on affiche simplement une variable, mais l'on pourrait également faire un calcul et afficher le résultat : `{{ 1 + 2 }}`. En revanche, AngularJS est exécuté côté client, contrairement à Django qui est côté serveur.

Comme précédemment, l'on a assigné une variable avec tous les articles. Cependant cette fois, la boucle se fait directement en utilisant la directive `ng-repeat` qui se confond avec la syntaxe HTML. A l'intérieur de l'attribut, il faut utiliser la syntaxe Angular pour faire la boucle : `article in articles`. On lui demande de parcourir la variable `articles` et d'utiliser comme variable temporaire `article` pour chaque article parcouru. Il existe beaucoup d'autre directives dans AngularJS, par exemple pour réagir au clic d'une souris sur un élément, pour afficher ou cacher des sections. Il est aussi possible de créer ses directives personnalisées avec le comportement désiré. Créer ses propres directives permet, soit d'avoir un code plus clair, soit d'éviter la répétition. Dans les deux cas, cette fonctionnalité est très utile et puissante.

A cause de ces directives, l'on parle d'HTML expressif. En effet, avec celles-ci, le HTML ne décrit pas seulement le contenu, mais aussi le comportement de l'application web et sa manière de fonctionner. L'on sait ainsi clairement et rapidement en regardant notre page HTML les fonctionnalités que l'on a implémentées sur celle-ci, ce que rend plus facile une vue d'ensemble de son application.

3.2.2 Two-way data binding

La deuxième fonctionnalité majeure d'AngularJS est ce que l'on appelle *two-way data binding* ou en français *la liaison des données à double sens*. Derrière cette mystérieuse expression se cache la manière qu'utilise le framework pour relier le modèle et la vue. Le système habituel est comme suit ; l'on génère les vues en fonction de ce qui se trouve dans le modèle, comme dans l'exemple précédent où l'on cherche des articles dans la *BD* pour ensuite générer une page. Lorsque le modèle change, un article est ajouté par exemple, la vue ne se met pas à jour. On doit la générer à nouveau pour voir le nouvel article. De plus, si l'utilisateur remplit un formulaire pour ajouter un nouvel article, le modèle ne change pas, tant que le formulaire n'a pas été traité. On appelle logiquement ce système *one-way data binding*. Le schéma qui suit illustre ce principe. Pour générer une vue pour l'utilisateur, le template et le modèle doivent être fusionnés et chaque fois qu'un changement est fait, l'on doit refaire le même processus.

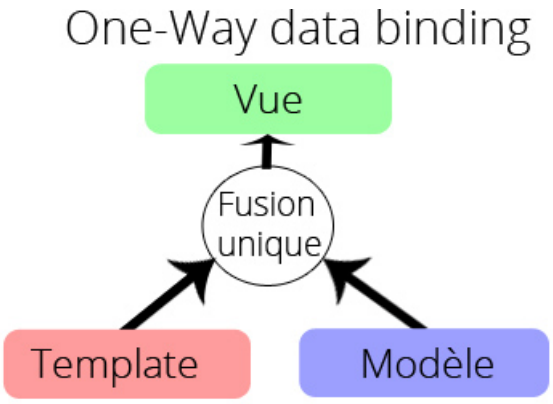

FIGURE 3.1 – One-way data binding

Avec Angular, le principe est plus intelligent. Les vues se génèrent effectivement en fonction des modèles, en revanche, si le modèle change, la vue se met automatiquement à jour sans avoir effectué un nouveau rendu de la page. Si un utilisateur fait un changement dans la vue, le modèle se change également. Les deux entités sont donc toujours synchronisées grâce à ce mécanisme du framework. La vue met à jour le modèle et le modèle met à jour la vue, continuellement.

Cette fonctionnalité facilite énormément la vie du développeur. Imaginons un système de commentaires. Il y a une liste de commentaires et un formulaire pour en rajouter. Pour le développeur, il suffit de relier le formulaire au modèle. Ensuite, au fur et à mesure que l'utilisateur tape son commentaire, le modèle est mis à jour et contient

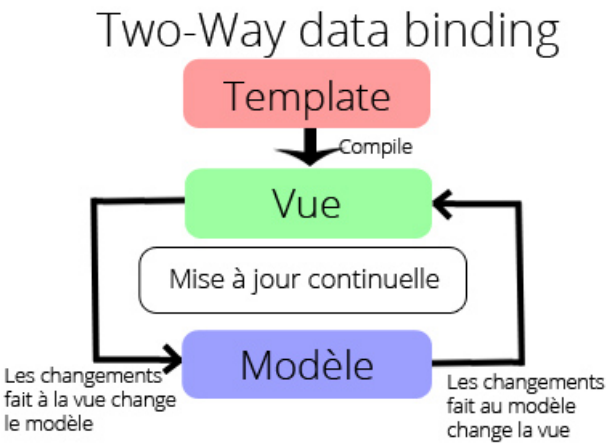

FIGURE 3.2 – Two-way data binding

le nouveau commentaire. Il peut déjà s'afficher dans la liste. Voici un exemple de code qui permet de cacher ou d'afficher une portion de page à l'aide d'un bouton.

```
# index.html
<html ng-app="DemoApp">
    <head></head>
    <body ng-controller="IndexController">
        <!-- section affichée selon la variable "affiche" grâce à la directive ng-show -->
        <div ng-show="affiche">
           <h1>Je suis une section cachée !</h1>
           <p>Mais je ne cache rien d'intéressant...<p>
        </div>
        <!-- bouton qui affiche/cache la section. Appelle la fonction toggle()
        grâce à la directive ng-click -->
        <button type="button" ng-click="toggle()">Afficher/Cacher</button>
    <body>
<html>

# index_controller.js
// On crée une application Angular
var app = angular.module("DemoApp");

// On crée un contrôleur Angular
app.controller("IndexController", function($scope) {

    // variable utilisée dans ng-show="affiche"
    $scope.affiche = false;

    // fonction appelée lorsqu'on clique sur le bouton
    $scope.toggle = function() {
        $scope.affiche = !$scope.affiche
    };

});
```

Tout d'abord, l'on affiche une section selon une variable booléenne `affiche` et l'on assigne à la variable la valeur `false` par défaut. La section est donc cachée. Puis l'on ajoute un bouton qui exécute une fonction qui change la valeur de notre variable `affiche` de `false` à `true` et vice-versa. La section s'affiche ou se cache selon son état lorsqu'on clique sur le bouton. Plusieurs directives sont utilisées dans l'exemple. `ng-app` signale à AngularJS qu'il faut analyser et compiler cette page. `ng-controller` signale qu'il faut utiliser le contrôleur `IndexController` qui est défini dans le fichier JavaScript et exécuter le code à l'intérieur. `ng-show`

montre ou non la section selon la contenu de la variable booléenne `affiche` et `ng-click` exécute la fonction `toggle()` lorsque que l'on clique sur le bouton.

3.2.3 Et plus encore...

Il y a évidemment encore d'autres avantages à utiliser ce framework, notamment les injections de dépendances et l'extensibilité d'AngularJS, mais nous avons vu les deux principales différences dans le monde des frameworks JavaScript.

CHAPITRE 4

Fonctionnalités

4.1 Les professeurs

Les comptes “professeurs” du site web ont la possibilité de rédiger des cours complets qui sont ensuite consultables par les autres utilisateurs du site, principalement des élèves. L’objectif est de pouvoir offrir du contenu théorique d’apprentissage en complément de partie pratique. Les enseignants disposent d’un outil formidable pour écrire des compléments au cours de base, afin d’approfondir des éléments ou de les clarifier pour les élèves en difficulté.

4.1.1 Création

La première étape du processeur de rédaction consiste à créer le cours en entrant les informations basiques du cours, à savoir le nom, la description, la catégorie et la difficulté. En ce qui concerne le nom et la description, ces champs parlent d’eux-mêmes. Il s’agit de saisir les informations pertinentes qui correspondent au contenu du cours afin que celui-ci puisse attirer les utilisateurs. Pour la catégorie, il faut choisir parmi les chapitres proposés, qui sont eux-mêmes regroupés par thèmes. Par exemple, un cours sur “les tangentes” se place dans le chapitre “Les cercles” qui lui-même est dans le thème “Géométrie”. Ainsi, les différentes ressources du site sont classées, et l’élève retrouve facilement la matière qui l’intéresse. Finalement, le professeur choisit la difficulté, sur une échelle de 1 à 3, de facile à difficile.

Tous les champs sont obligatoires. Une fois les champs dûment complétés, il suffit à l’auteur de cliquer sur le bouton “Envoyer” (1) pour être redirigé sur la page de rédaction où il peut commencer à écrire.

4.1.2 Rédaction

Structurer son cours

Pour structurer son cours, l’auteur peut tout d’abord créer plusieurs pages ayant chacune un titre (1). Ensuite dans chacune de ces pages, le contenu se découpe en plusieurs sections avec un titre (2) et un contenu (3). L’avantage de cette structure est que le site génère automatiquement un sommaire interactif du cours, en se basant sur le titre des pages et celui des sections.

L’action de créer des pages ou des sections et de modifier leur contenu se fait de manière claire, simple et surtout rapide. Le professeur travaille sans jamais devoir recharger sa page, action généralement lente. Pour ajouter une section, un bouton se trouve en bas de page (4). En cliquant dessus, une nouvelle zone d’édition va simplement apparaître. En bas de la page se trouvent le bouton (5) pour ajouter une nouvelle page, ainsi que la liste des différentes pages déjà créées (6). L’on peut donc, soit créer une page en cliquant sur le bouton (5), ce qui a pour effet d’afficher une nouvelle page vierge, soit alors naviguer entre les pages du cours pour éditer leur contenu en cliquant sur le numéro des pages (6). Il se peut aussi que l’on veuille réorganiser les sections dans un ordre différent, ou supprimer une section. Cette action est tout à fait possible ; à côté de chacune des sections se trouve trois boutons (7) : celui pour supprimer la section et les deux autres pour la monter ou la descendre. Lorsque l’on clique sur l’un des trois, la page est mise à jour automatiquement et instantanément.

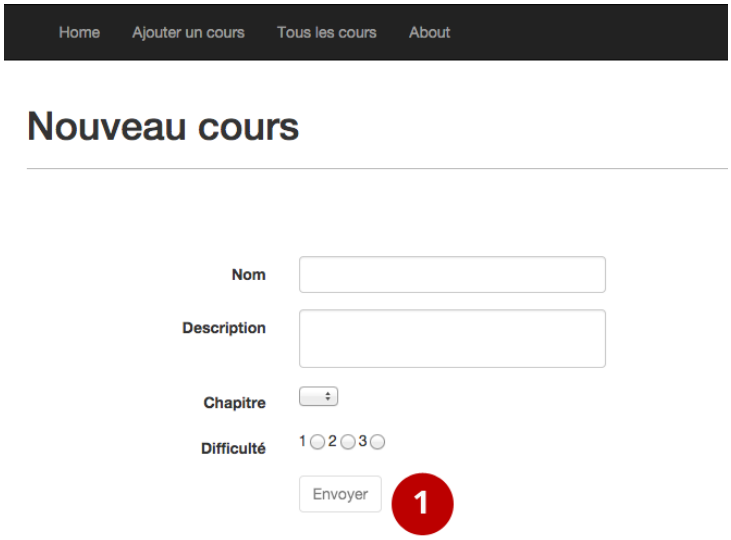

FIGURE 4.1 – Création d'un cours

En ce qui concerne l'enregistrement, il se fait automatiquement toutes les 30 secondes. De plus, au cas où l'on crée ou change une page, ou réorganise les sections, lorsque l'on prévisualise le cours, publie ou retire, de même que si l'on quitte la page, le cours est aussi sauvegardé. Il y a un bouton "Enregistrer" (8) en haut de la page pour faire une sauvegarde manuelle. L'heure du dernier enregistrement est affichée à la gauche du bouton.

Mise en forme

Pour mettre en forme le texte de son cours, le rédacteur utilise la syntaxe populaire *Markdown*. La syntaxe consiste à mettre dans le texte des marques avec des symboles qui sont ensuite interprétés, soit comme des titres, des images, du texte gras, etc. Par exemple, pour mettre en italique, on entoure un mot avec "*". Ainsi l'on peut facilement souligner du texte, ajouter des images, faire des tableaux et des listes, etc. S'il n'est pas encore à l'aise avec la syntaxe, le professeur peut à tout moment cliquer sur le bouton d'aide (9) en haut de la page, représenté par un point d'interrogation. Il découvre alors des liens pour avoir un rapide aperçu de l'ensemble des fonctionnalités du *Markdown*.

De plus, comme les cours visent avant tout un contenu mathématique, l'auteur peut (et doit) baliser le contenu mathématique. Ainsi celui-ci est formaté pour le rendu final et donc correctement affiché. La notation à utiliser pour les formules mathématique est celle du populaire format *LaTex*. Par conséquent, il suffit au professeur de mettre les balises et d'écrire ensuite sa formule mathématique en utilisant la syntaxe *LaTex* (10). Il y a deux types de balises. `\(` ... `\)` est utilisée pour intégrer des mathématiques directement dans le texte. La balise `||` est utilisée pour un affichage en bloc, c'est-à-dire que la formule est sur une nouvelle ligne, séparée du texte. A l'instar du *Markdown*, on peut se référer à la section d'aide pour se familiariser avec la syntaxe *LaTex* (9).

L'auteur n'ayant pas un aperçu direct du rendu de son texte, il peut cliquer sur le bouton en haut "Aperçu" (11) pour regarder comment le contenu est réellement affiché. Il n'a pas besoin de sauvegarder avant de faire l'aperçu. Une fois sur la page de prévisualisation, il peut retourner à la rédaction du cours en cliquant sur "Retour" en haut à gauche.

Ajouter des médias

Evidemment, l'auteur peut inclure des vidéos et des images pour enrichir son contenu. Il peut ajouter une image provenant d'internet en utilisant la syntaxe *Markdown* et le lien de l'image, tandis que pour la vidéo il peut en inclure une provenant de Youtube avec le même procédé que pour l'image (voir aide *Markdown*).

Modifier les informations de base

Lorsque l'enseignant désire modifier les informations saisies lors du processus de création, il peut cliquer sur le lien "Editer" (12) à côté du titre du cours. Il est alors sur une page avec le même formulaire qu'à la création du cours. L'auteur peut donc éditer le titre, la description, la catégorie et la difficulté et cliquer sur "Envoyer". Finalement, il est possible de retourner sur la page de rédaction en cliquant sur "Retour" en haut à gauche.

Publier un cours

Par défaut, le cours créé par un professeur n'est pas visible par les utilisateurs du site, afin de laisser à l'auteur le temps de rédiger le cours en entier. Lorsqu'il estime le cours prêt, il peut tout simplement cliquer sur le bouton "Publier" (13) tout en haut de la page pour rendre le cours visible. Il peut bien sûr retirer son cours de la liste à tout moment s'il désire apporter des modifications ou estime qu'il ne doit plus être lisible sur le site.

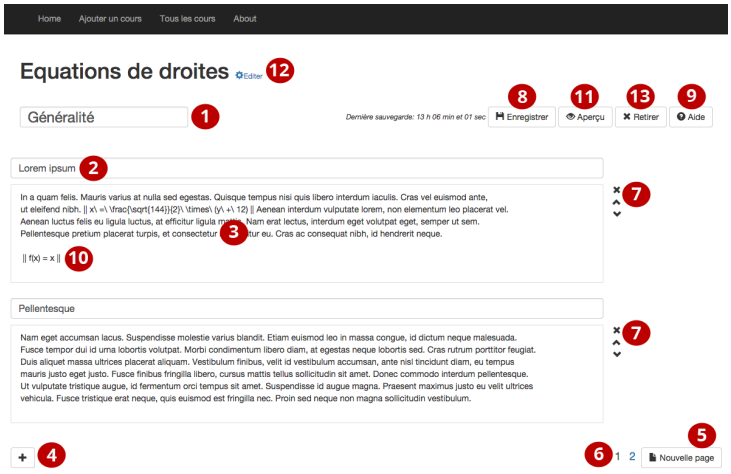

FIGURE 4.2 – Edition d'un cours

Retrouver son cours

En se rendant sur "Tous les cours" (1), le professeur voit la liste de tous les cours, qu'ils soient publiés ou non. S'il clique sur le titre d'un de ceux-ci (2), il retourne sur l'interface d'édition de son cours et peut y apporter les modifications désirées.

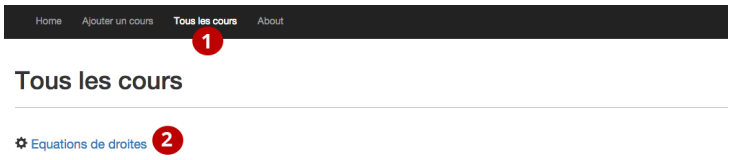

FIGURE 4.3 – Tous les cours des professeurs

4.2 Les étudiants

La plupart des fonctionnalités nécessitent d'être connecté au site.

4.2.1 Trouver un cours

Tous les cours publiés sont consultables par les utilisateurs du site. S'ils se rendent sur la page "Home", les étudiants ont la liste du contenu mis à leur disposition. Ils peuvent afficher tous les cours (1), seulement leurs favoris (2), ou alors trouver un cours par thème (3). Grâce à cette organisation, chaque élève peut trouver rapidement et efficacement le cours qui répond à ses besoins. Lorsqu'un cours l'intéresse, il lui suffit de cliquer dessus (4) et il est alors redirigé sur la page de lecture du cours.

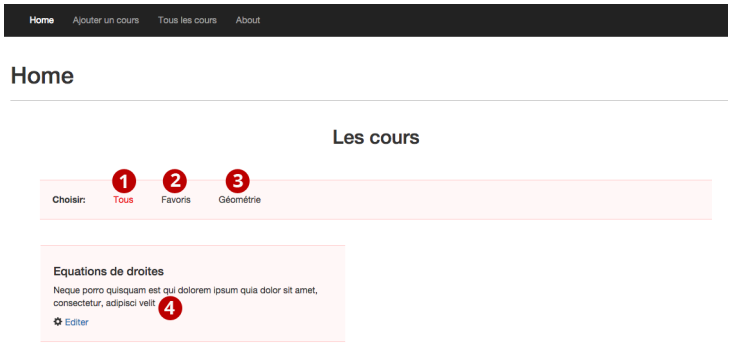

FIGURE 4.4 – Tous les cours

4.2.2 Lire un cours

Les favoris

S'il apprécie particulièrement un cours, le trouve utile ou veut le retrouver facilement par la suite, l'étudiant peut l'ajouter en favoris, à l'instar des favoris d'un navigateur par exemple. Pour ce faire, il y a une étoile en haut de chaque page (1) de lecture d'un cours. Elle est d'abord vide, ce qui signifie que le cours n'appartient pas aux favoris de l'étudiant. Si l'on clique dessus, l'étoile devient pleine et signifie que le cours est ajouté à la liste de favoris. A l'inverse, l'élève peut évidemment retirer un favori de la même manière qu'il l'a ajouté. Sa liste de favoris peut être retrouvée sur la page d'accueil comme expliqué précédemment.

La progression

L'élève profite également d'un système d'indication de sa progression. L'objectif est de pouvoir faciliter son apprentissage à travers le cours, l'aider à suivre et identifier ses zones de faiblesse. En bas de chaque page d'un cours se situent deux boutons intitulés "Compris" et "A relire" (2). A la fin de sa lecture de la page, il est conseillé à l'élève de cliquer sur l'un des deux boutons. En effet cela lui permettra ensuite de situer sa progression dans le cours. Basée sur les indications de l'utilisateur, une barre de progression en haut de la page (3) indique le nombre de pages comprises par rapport aux nombres de pages totale du cours. Quand il clique sur l'un des deux boutons, le lecteur est directement dirigé vers la page suivante et sa barre de progression est mise à jour. Il est évidemment possible de simplement cliquer sur "Suivant" (4) pour aller à la page suivante sans marquer la progression. Il y a aussi un bouton "Précédent" (5) pour se rendre à la page précédente.

Télécharger le cours

Il est souvent pratique de pouvoir lire un cours sans connexion internet ou de pouvoir l'imprimer pour le lire sans ordinateur. C'est pourquoi il est possible d'obtenir le cours en version PDF ! Lorsque l'étudiant lit un cours, il peut voir un lien "Télécharger le cours" dans le menu à droite (6). Il suffit de cliquer sur le lien et après quelques instants l'on obtient le PDF.

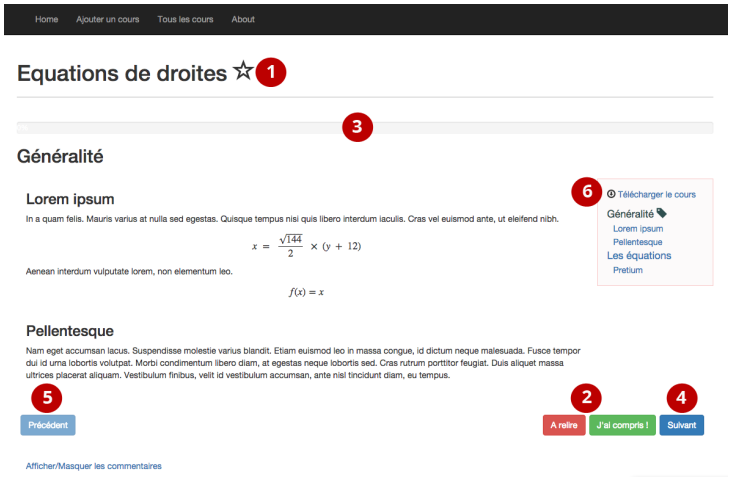

FIGURE 4.5 – Lire un cours

Les commentaires

Chaque lecteur a la possibilité de commenter un cours, pour le complimenter, émettre une critique ou poser une question. En bas de chaque page du cours se trouve un lien "Afficher/masquer les commentaires" (1). Cela permet de montrer tous les commentaires ou de les cacher en cliquant dessus. Dans cette section, l'utilisateur voit chaque commentaire du cours avec son auteur et la date de publication (2). En dessous des commentaires se trouve une zone de texte (3) dans laquelle l'on peut écrire son propre commentaire. Au fur et à mesure que l'on tape son message, le commentaire s'affiche dans la liste (4), mais n'est pas encore envoyé. Une fois le texte rédigé, il faut le poster en cliquant sur le bouton "Envoyer" (5) pour qu'il soit instantanément visible par les autres lecteurs.

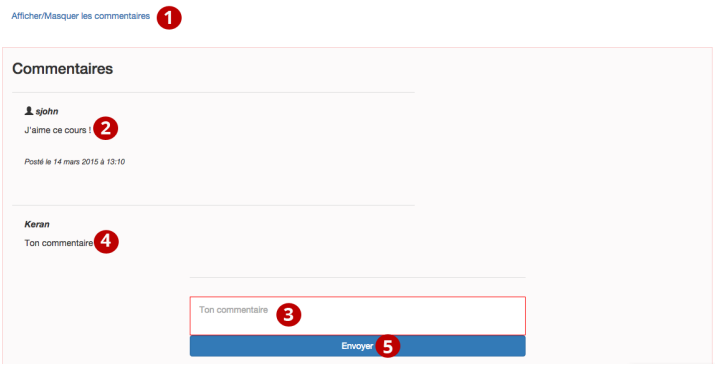

FIGURE 4.6 – Les commentaires

CHAPITRE 5

Schémas

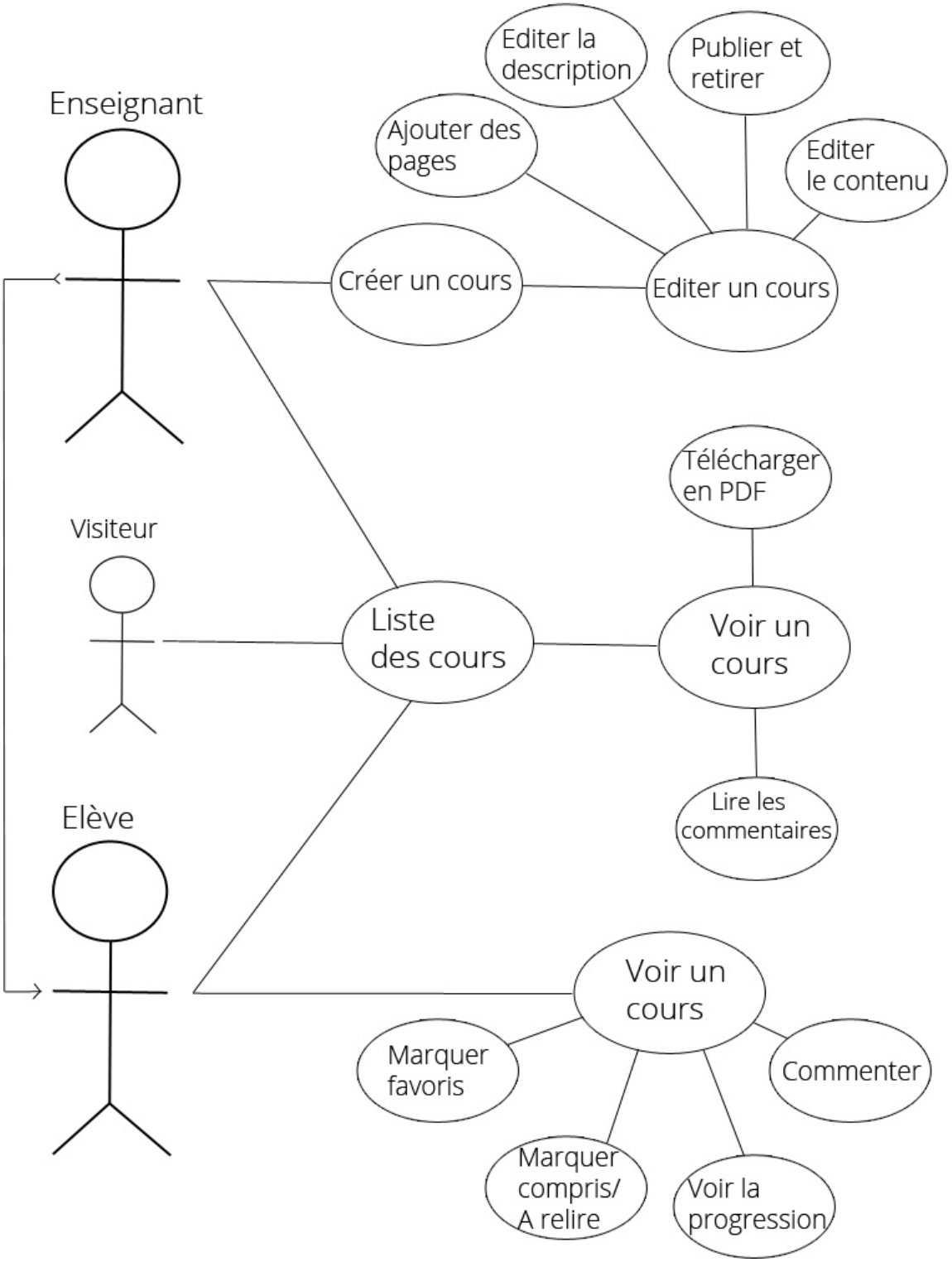

FIGURE 5.1 – Cas d'utilisation

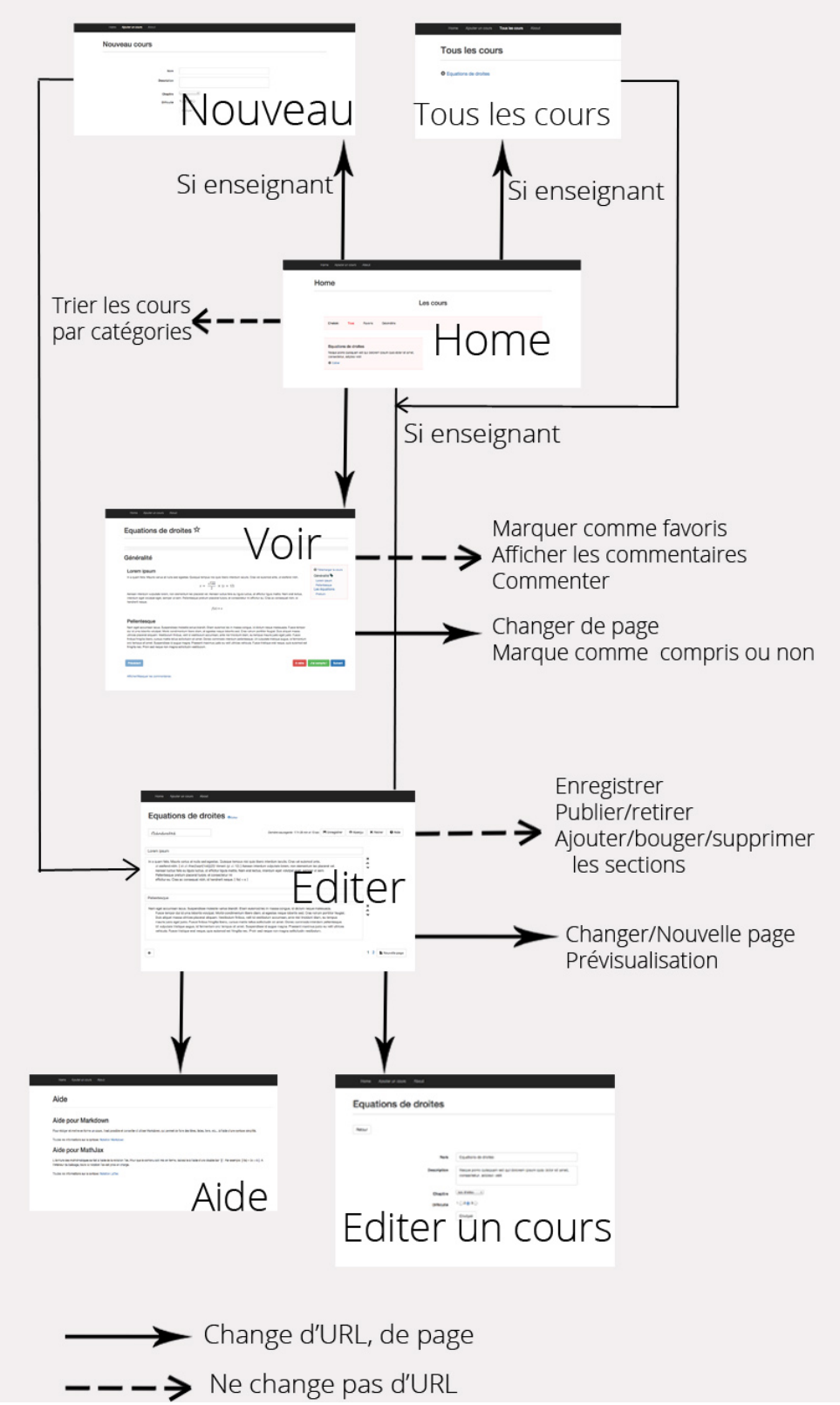

FIGURE 5.2 – Schéma de navigation

CHAPITRE 6

Modèle relationnel

6.1 Introduction

Une base de données est un outil permettant de stocker des données persistantes pour ensuite les réutiliser ou les conserver. Dans le domaine du web, on utilise principalement les bases de données relationnelles. Traditionnellement, elles se découpent en plusieurs tableaux que l'on appelle tables, contenant des colonnes et des lignes. On crée des tables pour représenter des entités, des cours par exemple, qui ont des attributs représentés par des colonnes. Ensuite, chaque ligne correspond à un enregistrement, c'est-à-dire une entité, un cours dans notre exemple. Une table a pratiquement toujours une colonne `id` qui est un nombre, un identifiant unique qui permet d'identifier un enregistrement parmi les autres de la table. On l'appelle *clé primaire*. Ils servent aussi à créer des relations entre les tables, à lier un enregistrement à un autre. Nous verrons ces relations dans la construction du module de cours. Pour communiquer avec une base de données relationnelles, notamment chercher des enregistrements dans une table, créer ou mettre à jour un enregistrement, etc. l'on utilise le langage *SQL*, *Structured Query Language*.

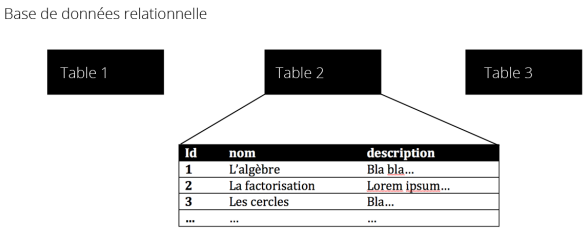

FIGURE 6.1 – Schéma résumant une base de données relationnelle

Le modèle relationnel est une modélisation de la base de données du site. Attardons-nous donc sur le modèle qui se cache derrière les fonctionnalités que nous voulons développer. Nous commencerons par le point central de la base de données : les tables et les relations qui concernent les cours et qui forment la majeure partie du modèle. Ensuite nous verrons les tables additionnelles qui complètent le modèle relationnel et ajoutent les fonctionnalités auxiliaires.

Il est important de savoir que Django fournit en tant que framework plusieurs outils facilitant le travail avec une base de données. Chaque table de notre *BD* est représentée par ce que l'on appelle un modèle. C'est un simple fichier Python qui contient les informations pour construire la table. Dans le projet, les modèles sont regroupés dans le fichier `models.py`. Ce fichier permet à Django de générer les tables et ensuite de fournir une série de méthodes qui permettent de communiquer avec la *BD* sans utiliser le langage SQL, qui est le seul langage que comprend une *BD*. Django nous évite par conséquent d'apprendre un nouveau langage. Nous verrons ces méthodes plus tard dans les exemples d'utilisation.

6.2 Les cours

6.2.1 La structure

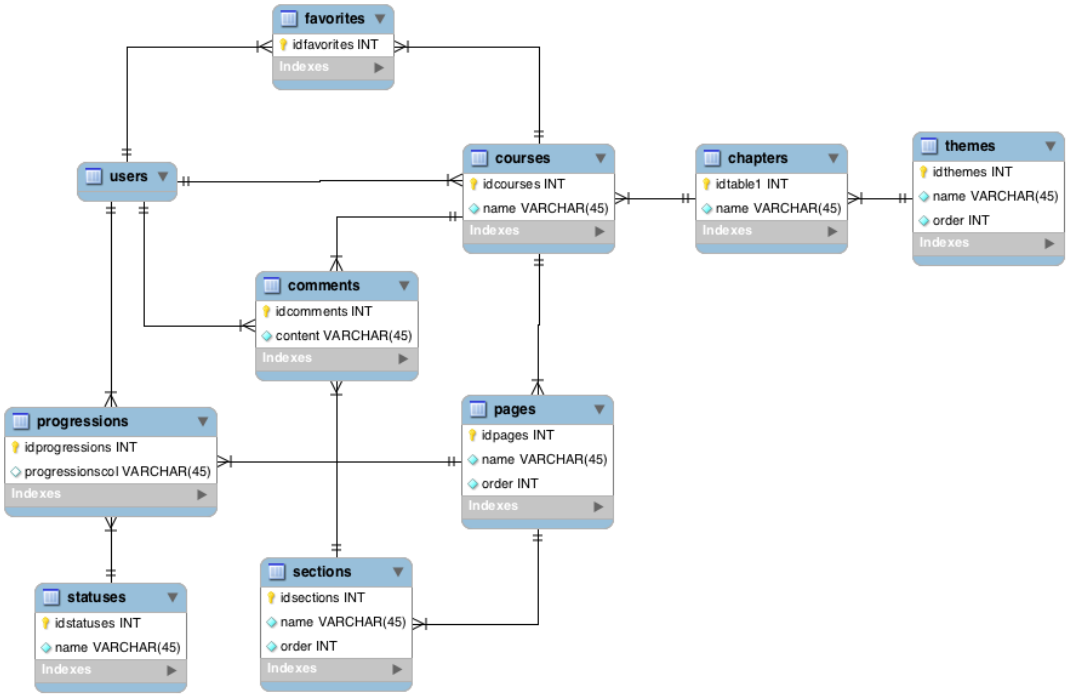

FIGURE 6.2 – Schéma de toutes les tables du modèle relationnel

La structure des tables relationnelles reflète la structure que perçoit le rédacteur et cela facilite grandement la compréhension. Nous avons vu qu'un cours se compose de plusieurs pages. Elles-mêmes contiennent plusieurs sections, avec un titre et un contenu, que l'auteur peut éditer ou retirer à sa guise. C'est exactement la même chose dans la structure du modèle. Tout d'abord on trouve une tables `courses` qui contient les informations de base que le professeur entre à la création du cours. Elle contient les colonnes suivantes : `name`, `description`, `difficulty`. Il reste le champ `chapter` que nous verrons plus tard. Ensuite, il y a la table `pages` avec la colonne `name`, `order` et `course_id`. `name` est le titre de la page. `order` est un nombre qui permet de trier les pages d'un cours entre elles et de les réorganiser par la suite. `course_id` indique la relation avec un cours. Elle contient l'`id` du cours auquel appartient la page. En effet, chaque page appartient à un cours et, vice-versa, un cours possède donc plusieurs pages. L'utilisation plus précise de cette relation est expliquée dans la partie suivante. Finalement, pour compléter l'ensemble, il reste la table `sections`. Elle possède les colonnes `name`, `content`, `order` et `page_id`. On retrouve un titre et un contenu. Il y également l'ordre qui fonctionne comme pour les pages. `page_id` indique la relation avec une page. Elle contient l'`id` de la page à laquelle appartient la section. Par conséquent, une section appartient à une page et une page possède plusieurs sections. A noter que ces trois tables ont également les champs `created_at` et `updated_at`. Elles enregistrent la date et l'heure de la création et la dernière mise à jour de l'entité. Pour résumer les relations qui lient les tables, un cours a plusieurs pages, et chacune des pages a plusieurs sections. Comme dit précédemment, l'on comprend facilement les relations en observant l'implémentation de l'interface de rédaction d'un cours.

6.2.2 Utilisation

Tous les exemples d'opérations sur la base de données sont d'abord écrits avec les méthodes de Django puis en SQL pur. Une des particularités de Django pour sauvegarder des objets dans la base de données est qu'il fait appel à des formulaires. Ils sont nommés dans le code sous la forme de ....Form, comme `CourseForm` par exemple. On utilise aussi ces formulaires dans les vues pour générer les formulaires HTML que complètent les utilisateurs. Concrètement, ils permettent simplement de relier les données soumises par un utilisateur à nos modèles Django.

FIGURE 6.3 – Schéma qui résume les relations des tables courses, pages et sections

Rappelons que dans Django les modèles sont la représentation des tables de la *BD*. Par conséquent les formulaires Django servent à créer et à mettre à jour des enregistrements de la base de données via des formulaires HTML. Prenons un exemple pour bien se représenter le processus. Lorsqu'un utilisateur soumet un formulaire, le navigateur envoie les données au serveur sous la forme d'un dictionnaire : `{"titre" : "La géométrie", "description" : "Bla bla"}`. Ensuite, côté serveur, l'on récupère le dictionnaire et enregistre les données dans la *BD* en utilisant un formulaire Django. Pour approfondir le concept des formulaires, il faut se rendre sur la documentation Django [1].

Récupère tous les cours.

```
# api.py - TeacherCourseList

Course.objects.all()

SELECT * FROM courses
```

Récupère tous les cours publiés ayant un thème particulier.

```
# api.py - CourseList

Course.objects.filter(chapter__theme__name=request.GET['theme'], published=True)

SELECT * FROM "courses_course" INNER JOIN "teachers_chapter" ON ( "courses_course"."chapter_id" =
INNER JOIN "teachers_theme" ON ( "teachers_chapter"."theme_id" = "teachers_theme"."id" )
WHERE ("teachers_theme"."name" = "Géométrie" AND "courses_course"."published" = True)
```

Créer un nouveau cours. On crée d'abord le cours, puis une page associée contenant une section vierge.

```
# api.py - CourseList

# on utilise un formulaire (CourseForm)
# request.data est un dictionnaire contenant les données soumises par l'utilisateur
# ici les informations du cours
course_form = CourseForm(request.data)
# on vérifie si les informations sont présentes et valides
if course_form.is_valid():
    # on crée le cours
    course = course_form.save()
    # on crée la page associée
    page = Page(name="Première page", order=1, course_id=course.id)
    page.save()
    # on crée une section associée à la page
    page.sections.create(name="Première section", order=1)

-- on crée le cours
INSERT INTO courses (name, description, difficulty, author_id, chapter_id, created_at, updated_at
VALUES ("L'algèbre", "Lorem ipsum...", 3, 1, 1, *, *)
-- => ID du cours = 1
```

1. https://docs.djangoproject.com/fr/1.7/topics/forms. Consulté le 20 décembre.

```
-- On crée la page associée
INSERT INTO pages (name, order, course_id, created_at, updated_at)
VALUES ("Première page", 1, 1, *, *)
-- => ID de la page = 1
-- on crée une section associée à la page
INSERT INTO sections (name, content, order, page_id, created_at, updated_at)
VALUES ("Première section", "bla bla", 1, 1, *, *)
```

Mettre à jour le contenu d'une page d'un cours. Le titre de la page, le titre et le contenu des sections vont être sauvegardés. Pour accomplir cette action, on commence par simplement enregistrer la page avec les nouvelles données. On utilise la même procédure que pour la création d'un cours. Remarquons simplement que dans `page_form = PageForm(request.data, instance=page)`, l'on passe en paramètre la page provenant de la base de données, pour signaler à Django que l'enregistrement existe déjà. Ainsi Django ne crée pas une nouvelle page, mais met la nôtre à jour. Pour les sections, on itère d'abord sur le dictionnaire qui contient les données de toutes les sections de la page. Puis, pour chaque section, on accomplit la même procédure que pour une page.

```
# api.py - PageCourseDetail

# Récupère la page à éditer
page = Page.objects.get(id=page_id)
# On utilise un formulaire (PageForm)
# request.data est un dictionnaire contenant les données soumises par l'utilisateur
# ici le contenu de la page
page_form = PageForm(request.data, instance=page)
# On vérifie si les informations sont présentes et valides
if page_form.is_valid():
    # On enregistre la page - sauvegarde le titre
    page_form.save()
# On récupère le dictionnaire contenant les données des sections
sections_params = request.data['sections']
# On fait une boucle pour chaque section
for section_params in sections_params:
    # On récupère la section
    section = Section.objects.get(id=section_params['id'])
    # On utilise un formulaire (SectionForm)
    # section_params est un dictionnaire contenant le titre et le contenu de la section
    section_form = SectionForm(section_params, instance=section)
    # On vérifie si les informations sont présentes et valides
    if section_form.is_valid():
        # On enregistre la section
        section_form.save()
```

```
-- Récupère la page à éditer
SELECT * FROM pages WHERE id = 1
-- On enregistre la page
UPDATE pages SET name = "Nouveau titre" WHERE id = 1
-- On enregistre la section
UPDATE sections SET name = "Nouveau titre", "content" = "Lorem ispum" WHERE id = 1
```

6.3 Les chapitres

Pour pouvoir organiser le contenu du site, chaque cours est associé à un chapitre. Deux tables servent cet objectif. Tout d'abord il y a la table `themes` avec un champs `name`. Il y a également la table `chapters` avec un champ `name` et `theme_id`. `theme_id` associe chaque chapitre à un thème. Ensuite la table `courses` a un champ `chapter_id`. Celui-ci contient l'`id` d'un chapitre. Il relie chaque cours à un chapitre et par conséquent à un thème. Par exemple, il peut y avoir un cours sur les tangentes. On le placerait dans le chapitre "les cercles" et le chapitre se trouverait lui-même dans le thème "Géométrie". On peut légitiment se demander pourquoi ces deux niveaux et ces deux tables ? Le système est construit afin de laisser une plus grande souplesse et liberté pour

organiser le contenu. En effet, imaginons qu'il y ait 10, 20, 30 ou plus chapitres, comment s'y retrouver ? La solution est de les regrouper sous une idée plus générale, et c'est précisément le rôle de la table `themes`.

6.4 Les commentaires

La table `course_comments` permet aux lecteurs du site de poster un commentaire sur un cours. La table contient un champ `content`, `user_id` et `course_id`. Chaque commentaire appartient donc à un utilisateur et à un cours.

6.5 La progression

L'utilisateur a la possibilité de marquer sa progression quand il lit un cours. Voyons comment cette fonctionnalité se traduit au niveau du modèle relationnel. `progressions` est la table principale. Elle contient les colonnes `page_id`, `user_id` et `status_id`. En somme, elle ne contient que des relations. L'idée principale est la suivante ; lorsqu'un utilisateur a lu une page d'un cours, on lui propose de choisir s'il a compris ou souhaite relire la page. Le champ `user_id` enregistre quel utilisateur indique sa progression et le champ `page_id` indique quelle page est concernée. Finalement, l'attribut `status_id` associe la progression à une table `statuses`. Celle-ci contient le nom que peut avoir une progression. Il y a deux statuts : "Compris" et "A relire". Pour résumer, lorsque que l'on crée une progression dans notre base de données, l'on sait qu'un certain utilisateur a "compris" ou souhaite "relire" une page particulière. L'exemple qui suit montre comment l'on enregistre une progression concrètement.

```
# api.py - CoursePageProgress

# On récupère l'utilisateur connecté au site
user = request.user
# On récupère la page concernée
page = Page.objects.get(id=pk)

# request.data est un dictionnaire contenant les données soumises par l'utilisateur
# ici, si l'utilisateur a compris ou non la page
# On choisit le status en fonction
if request.data['is_done'] == True:
    status = Status.objects.get(name="Compris")
else:
    status = Status.objects.get(name="Relire")

# Si l'utilisateur n'a pas encore marqué sa progression sur cette page
if not page.state(user):
    # On crée une progression avec la page, le statut et l'utilisateur
    page.progression_set.create(status=status, user=user)
# si l'utilisateur a déjà marqué sa progression sur cette page
else:
    # On récupère sa progression
    progression = page.progression_set.get(user=user)
    # On met à jour avec le nouveau statut
    progression.status = status
    progression.save()

-- Récupère la page à éditer
SELECT * FROM pages WHERE id = 1
-- => ID de la page = 1
-- Récupère le statut
SELECT * FROM statuses WHERE name = "Compris"
-- => ID du statut = 1
-- Crée une progression
INSERT INTO "progressions" ("page_id", "status_id", "user_id", "created_at", "updated_at")
```

```
VALUES (1, 1, 1, *, *)
-- Met à jour une progression
UPDATE "progressions" SET status_id = 1 WHERE id = 1
```

CHAPITRE 7

RestLess

7.1 Présentation

RestLess [1] est un set d'outils permettant de faciliter l'implémentation d'une API JSON dans Django. Il a l'avantage d'être léger et facile à utiliser comme nous le verrons par la suite. Une API, Application Programming Interface, est basiquement une application qui offre des services accessibles par une autre application. Le JSON est un format de données dans le style d'un dictionnaire `{"nom" : "Keran", "prenom" : "Kocher"}`. On appelle donc une API JSON une application qui fournit des données en format JSON. Concrètement, il s'agit d'une série d'URL qui fournissent le contenu de différentes tables de la BD en format JSON. Quand on se rend sur une de ces URL, on ne voit pas une page HTML, mais simplement un dictionnaire de données. Sur notre site web rendez-vous sur `http://webmath.com/courses/api/themes` pour voir à quoi ressemble la page.

Dans la présentation d'AngularJS, nous avons vu que le framework n'est pas capable de communiquer directement avec une base de données. C'est une limitation de JavaScript, qui s'exécute du côté du client dans notre cas. Pour palier à ce problème, on utilise un langage intermédiaire qui est capable de communiquer avec une BD et qui s'exécute côté serveur, c'est le cas de Python et son framework Django. On va donc construire une API avec Django. API à laquelle notre application AngularJS va pouvoir accéder. Dans les fait, Django va chercher les données dans la base de données, les transforme en format JSON puis les sert. Il suffit ensuite avec AngularJS d'accéder à l'URL correspondante et l'on dispose ensuite des données que l'on peut utiliser à sa guise dans nos vues AngularJS.

7.2 Les vues génériques

Lorsqu'on développe une application web, cela donne souvent un code redondant, répétitif. En effet, quand on crée des fonctionnalités, il s'agit généralement d'une table dans la base de données, avec laquelle on communique pour ajouter ou modifier des données. Et ces interactions se ressemblent dans la plupart des cas. On parle généralement du CRUD : create, read, update and delete, en français créer, lire, mettre à jour et supprimer. On a une table et on veut accomplir les opérations CRUD dessus. Pour ce faire, l'on a une série d'URL qui exécutent différentes actions. Prenons l'exemple de la création d'un blog tout à fait typique. On va créer une table `articles`, et implémenter les actions suivantes : on veut pouvoir afficher tous les articles, ou n'en afficher qu'un seul, ajouter un nouvel article, le mettre à jour ou le supprimer. Ainsi, avec ces quatre opérations, l'on peut disposer d'un blog complet et fonctionnel. Pour beaucoup de fonctionnalités, il s'agit d'effectuer toujours ces mêmes opérations classiques ; un autre exemple serait un système de commentaires. Pour programmer ces outils basiques et conventionnels, deux moyens sont à notre disposition dans Django. D'abord, l'on peut programmer les opérations de A à Z. Avec Django, il s'agit basiquement de créer une URL et de lui assigner une fonction qui s'occupe de communiquer avec la base de données, et qui retourne une page HTML. Si l'on utilise cette méthode, on risque de devoir programmer souvent le même code au fil du développement et de perdre du temps. Ci-dessous le code pour afficher tous les articles avec la première méthode.

1. https://github.com/dobarkod/django-restless. Consulté le 04 javier 15.

```
# url.py - fichier qui gère les URL du site

from django.conf.urls import patterns, url, include

from courses import views

# On crée une URL /articles qui utilise la fonction index - voir fichier views.py
urlpatterns = patterns('',
    url(r'^articles$', views.index),
)

# views.py - fichier qui contient les fonctions liées aux URLs
# = vue ou contrôleur

from django.shortcuts import render

# Fonction reliée à /articles
def index(request):
    # Récupère tous les articles de la BD (fait appelle au modèle Article)
    articles = Article.objects.all()
    # Retourne le code HTML en utilisant le ficher courses.html
    return render(request, "courses/courses.html", locals())
```

La deuxième méthode consiste à utiliser les vues génériques. Ce sont des classes dans Django contenant les opérations conventionnelles déjà écrites. Il nous suffit donc de créer notre propre classe qui hérite d'une vue générique Django et d'ensuite la relier à notre URL, comme on le faisait précédemment avec la fonction. Pourquoi créer une classe et ne pas utiliser directement les classes Django ? On agit ainsi tout simplement pour pouvoir personnaliser la classe. Il faut déjà obligatoirement spécifier le modèle que doit utiliser la classe, par exemple pour savoir quels enregistrements elle doit aller récupérer. Regardons la même fonctionnalité qu'avant, mais écrite avec les vues génériques.

```
# url.py - fichier qui gère les URL du site

from django.conf.urls import patterns, url, include

from courses.views import ArticlesList

# On crée une URL /articles qui utilise la vue générique ArticlesList - voir fichier views.py
urlpatterns = patterns('',
    url(r'^articles$', ArticlesList.as_view()),
)

# views.py - fichier qui contient les fonctions liées aux URL
# = vue ou contrôleur

from django.views.generic import ListView

# La classe générique reliée à /articles
# Hérite de ListView, classe provenant de Django
class ArticlesList(ListView):

    # On spécifie le modèle à utiliser
    model = Article
```

Avec la seconde méthode, le code est plus concis. L'exemple montre comment générer la liste des articles, mais il existe une vue générique pour chaque opération CRUD. Il est encore possible de personnaliser notre classe `ArticlesList` avec des options ou en surchargeant les méthodes. En revanche, si notre fonctionnalité a des besoins spécifiques qui s'éloignent trop de la convention, les vues génériques ne sont plus adaptées, car leur personnalisation a évidemment des limites. Dans ces cas-ci, on retourne à la première méthode. Toutes les explications et options des vues génériques se trouvent dans la documentation Django [2].

2. https://docs.djangoproject.com/fr/1.7/topics/class-based-views/generic-display. Consulté le 04 javier 15.

7.3 Fonctionnement de RestLess

Nous avons étudié ce qu'étaient les vues génériques dans Django, parce que RestLess se base exclusivement sur ce concept pour construire une API JSON. En fait, RestLess fournit également des vues génériques qui sont des dérivées des classes Django. Les classes de RestLess font en effet exactement le même travail que celle de Django, à la différence qu'elles travaillent avec le format JSON. Ainsi, on peut construire facilement et rapidement notre API, en économisant du code et du temps. En revanche, Django possède beaucoup de vues génériques et RestLess n'offre que les plus utiles. Avant de nous intéresser aux classes que nous pouvons utiliser avec RestLess, il nous faut d'abord voir les différents types de requêtes qui existent dans le monde du web.

7.3.1 HTTP

HTTP est l'abréviation de *HyperText Transfer Protocol* qui veut dire *protocole de transfert hypertexte*. Ce protocole est utilisé sur internet pour la communication entre un client et un serveur. Le serveur est un ordinateur dont le rôle est de fournir le contenu désiré d'un site web. Le client est un navigateur utilisé par une personne qui navigue sur un site web. Lorsqu'un utilisateur visite une page, le navigateur demande au serveur la page HTML correspondante et ensuite il l'affiche à l'utilisateur. Pour établir le transfert de données, on utilise donc HTTP. Quand le client demande une information au serveur, on appelle cela une requête HTTP. Il y a plusieurs types de requêtes HTTP. Nous avons vu que le serveur envoie des données au client, mais le contraire est aussi vrai. Le client peut envoyer des données au serveur, quand il soumet un formulaire HTML par exemple. Ces différentes requêtes, formulées par un navigateur qui est le client, servent en général à agir sur une ressource en permettant notamment les opérations CRUD. On appelle une ressource une entité modifiable, souvent un enregistrement provenant d'une *BD*. La liste qui suit présente les différentes requêtes les plus importantes dans notre cas.

- GET : requête la plus courante, le serveur envoie les données au client, une page HTML par exemple. Aucune ressource modifiée.
- POST : le client envoie des données au serveur, souvent via un formulaire HTML. Le résultat est la création d'une ressource.
- PUT : le client envoie des données au serveur. Le résultat est la modification d'une ressource.
- DELETE : supprime une ressource.

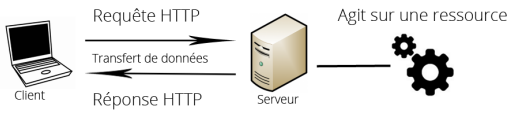

FIGURE 7.1 – Schéma de la communication entre un client et un serveur

Nous devons utiliser ces requêtes lorsqu'il s'agit de modifier nos ressources, c'est-à-dire les enregistrements de notre base de données. Par exemple on crée un cours, on le modifie ou on le supprime. Le travail de RestLess est de supporter ces requêtes. En clair, il doit fournir une URL et une fonction qui s'occupent de traiter les différents types de requêtes. Attention à ne pas confondre, une requête n'agit pas directement sur une ressource, c'est le serveur qui s'en occupe. La requête consiste juste en un transfert de données entre le client et le serveur et ainsi elle déclenche des actions.

7.3.2 Les classes RestLess

Maintenant que les bases sont en place, nous pouvons enfin nous intéresser à la liste des classes RestLess utilisées dans le projet avec les requêtes supportées et leur utilité.

- ListEndpoint
  - get : retourne toutes les ressources
  - post : crée une nouvelle ressource

- DetailEndpoint
  - get : retourne une ressource
  - put : met à jour la ressource
  - delete : supprime la ressource
- Endpoint
  - pour créer des actions spécifiques

La liste ci-dessus regroupe les trois vues génériques dont nos classes peuvent hériter. Elles permettent de réaliser les quatre opérations sur nos ressources ainsi que des actions personnalisées. La différence entre la classe `DetailEndpoint` et `ListEndpoint` est que la première agit sur une ressource particulière. Elle a donc besoin d'un identifiant dans l'URL pour savoir quelle ressource elle doit modifier, dans le style `/courses/:id`. Notons qu'évidemment toutes les actions, le code qui s'exécute derrière une URL, retournent du JSON. En effet, il s'agit de la particularité et de l'utilité de RestLess. Comment concrètement utiliser ces classes dans le projet ? On fait comme précédemment dans l'exemple sur les vues génériques Django. La première étape consiste à créer une classe qui hérite soit de `ListEndpoint`, de `DetailEndpoint` ou de `Endpoint`. La seconde étape consiste à spécifier le modèle. Ensuite il faut créer une URL dans laquelle l'on spécifie qu'il faut utiliser notre classe précédemment déclarée. Ainsi. quand on fait une requête sur cette URL, suivant le type de requête, Django fait appel aux méthodes provenant des vues génériques RestLess. Par exemple, on crée une classe `CoursesList` dans laquelle on spécifie le modèle `Course`. Ensuite on rattache cette classe à l'url `/courses`. Si l'on fait une requête de type POST sur `/courses`, Django va chercher la méthode `post`, même nom que le type de requête, dans la classe `CoursesList`. Comme il ne la trouve pas dans notre classe, il la cherche dans la classe parente `ListEndpoint` et l'exécute. On appelle ce principe l'héritage. Le résultat est qu'une ressource est créée dans la table `courses` avec les paramètres du client. Une réponse en JSON contenant la ressource créée est retournée. On peut également faire une requête GET sur `/courses` et le serveur retourne tous les cours en format JSON également. Rien n'est plus nécessaire pour avoir notre API JSON fonctionnelle.

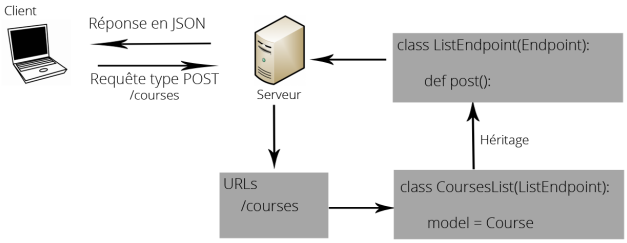

FIGURE 7.2 – API : schéma du traitement d'une requête à l'aide des vues génériques RestLess

Dans le cas de notre projet, il a fallu personnaliser les vues génériques pour répondre aux besoins spécifiques des ressources. Pour ce faire, il faut surcharger les méthodes héritées des classes RestLess. Comme mentionné précédemment, les méthodes ont le nom de la requête à laquelle elles correspondent. Si l'on fait une requête PUT, la méthode `put` est appelée et ainsi de suite. C'est un principe des vues génériques Django. Concernant les vues génériques, si notre classe ne contient pas le méthode appelée par une requête, le framework va automatiquement chercher la méthode dans la classe parente. Celle-ci est la classe RestLess qui contient les méthodes classiques et conventionnelles qui nous évitent un code redondant. En revanche, si l'on ne veut plus utiliser ces méthodes classiques car elles ne sont plus adaptées, l'on crée alors une méthode de même nom dans la classe fille. Cette méthode est desormais appelée à la place de celle de la classe mère. On appelle cela surcharger une méthode. Dans le projet, toutes les vues génériques sont écrites dans le fichier `api.py`. On peut y observer que plusieurs méthodes de tous types ont été surchargées. Il faut s'inspirer des méthodes RestLess que l'on surcharge pour que la nouvelle méthode accepte les bons arguments et retourne une réponse valide. On retrouve le fichier source sur la documentation RestLess [3].

On doit parfois retourner des objets JSON personnalisés, c'est-à-dire pouvoir choisir les paires clé/valeur de notre dictionnaire. Par défaut RestLess retourne simplement tous les attributs de l'enregistrement en question. On accomplit cette personnalisation généralement dans le but de choisir certains attributs, d'en créer des nouveaux qui ne sont pas des champs de la table ou de joindre des enregistrements associés. RestLess fournit la méthode

3. https://django-restless.readthedocs.org/en/latest/_modules/restless/modelviews.html. Consulté le 04 javier 15.

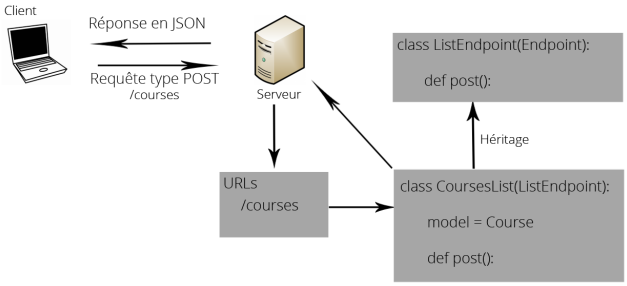

FIGURE 7.3 – Surcharge d'une méthode

`serialize` pour résoudre ce problème. Par exemple, pour un cours nous avons besoin de joindre les pages associées et leur contenu ainsi que le nombre total de pages. On peut se rendre sur la documentation [4] pour plus d'informations et sur le fichier `api.py` pour des exemples d'utilisation.

7.3.3 Communication avec l'API

Nous avons construit une API JSON afin qu'AngularJS puisse communiquer avec une base de données. Pour effectuer les requêtes sur l'API, on trouve deux méthodes utilisées dans le projet. La première consiste à utiliser l'objet Angular `$http`. Celui-ci permet de construire une requête et de récupérer la réponse ainsi que les éventuelles erreurs. On trouve toutes les spécifications sur la documentation AngularJS [5]. La seconde méthode est d'utiliser l'objet `$resource`. Si l'on possède une table sur laquelle on veut effectuer les opérations CRUD, `$resource` nous évite d'écrire toutes les requêtes avec `$http`. En effet, `$resource` est un objet qui fournit directement les méthodes pour effectuer les différents types de requêtes sur l'API. Pour générer les URL, on fournit d'abord à l'objet une URL de base. Puis, en se basant sur les conventions du web, l'objet est capable d'effectuer les requêtes servant à agir sur la table. Ci-dessous se trouve un exemple d'un objet `$resource` et de son utilisation partielle.

```
var Section = $resource(
    "api/sections/:sectionId"
);

Section.query();
// => GET /api/sections
var section = Section.get({sectionId: 1});
// => GET /api/sections/1
card.$save;
// => POST /api/sections/1
```

Dans notre application, tous les objets `$resource` sont définis dans le fichier `/courses/static/courses/javascripts/factories/resources.js`. La documentation **[#f5]_** fournit toutes les explications concernant `$resource`.

4. https://django-restless.readthedocs.org/en/latest/#. Consulté le 04 javier 15.
5. https://docs.angularjs.org/api/ng/service/$http. Consulté le 21 mars 15.

CHAPITRE 8

MathJax

MathJax [1] est un plugin JavaScript pour afficher des mathématiques dans le navigateur. Il utilise HTML et CSS pour afficher les expressions de mathématique dans le style LaTeX. Le rédacteur peut par conséquent aisément utiliser la notation LaTeX pour définir des fonctions, faire des fractions, des racines carrées, des puissances, écrire les symboles spéciaux, etc. en balisant le contenu et ensuite celui-ci sera correctement affiché grâce au moteur de rendu MathJax. Une section d'aide est disponible pour la syntaxe à utiliser.

8.1 Installation

L'installation de MathJax est très facile sur n'importe quelle site web. Il suffit de télécharger le dossier sur le site officiel [1], de le mettre dans le projet et d'inclure le lien du fichier JavaScript `MathJax.js` à la racine du dossier. Il est possible de charger une configuration différente de celle par défaut, pour changer la mise en forme, la notation et d'autres subtilités. On peut passer le nom de la configuration dans un paramètre `config` dans le lien qui inclut MathJax. Tous ces fichiers se trouvent dans le dossier `config` de MathJax. Il est aussi possible de créer sa configuration personnalisée. Les informations détaillées des configurations disponibles se trouvent dans leur documentation [2]. Actuellement nous utilisons la configuration `TeX-AMS_HTML` qui utilise la notation Tex (LaTeX) et qui génère du HTML. Il est aussi possible d'utiliser le CDN. On peut également spécifier dans le lien le fichier de configuration.

Une fois le plugin MathJax installé, il va automatiquement scanner les pages HTML et détecter les balises qui délimitent du contenu mathématique, par défaut `$$`. Ensuite il va mettre en forme le contenu afin qu'il s'affiche correctement. Ci-dessous se trouve un exemple de l'intégration du plugin et du résultat.

```
<html>
    <head>
        <!-- Fichiers locaux -->
        <script src="/MathJax/MathJax.js?config=TeX-AMS_HTML"></script>

        <!-- CDN -->
        <script src="http://cdn.mathjax.org/mathjax/latest/MathJax.js?config=TeX-AMS_HTML"></scri
    </head>
    <body>
        $$ f(x) = 5x + 4 $$
    </body>
</html>
```

$$f(x) = 5x + 4$$

FIGURE 8.1 – Résultat de MathJax

1. http://mathjax.org. Consulté le 27 décembre.
2. http://docs.mathjax.org/en/latest/config-files.html. Consulté le 27 décembre 14.

8.2 Personnalisation

Dans le cadre du projet, quelques modifications ont été faites par rapport à l'utilisation basique de MathJax. Pour pouvoir le personnaliser, on utilise un fichier spécial pour modifier les options que MathJax met à disposition. Ce fichier est déjà présent dans le dossier. Il se trouve dans le sous-dossier `config/local` et se nomme `local.js`. Ensuite, lorsque l'on inclut MathJax dans notre page, il suffit de modifier l'URL comme suit : `config=TeX-AMS_HTML,local=local`. On a ajouté dans le paramètre `config` de l'URL `,local/local`.

Etudions ce fichier qui a été modifié pour les besoins du projet.

```
MathJax.Hub.Config({

    skipStartupTypeset: true,

    // Apparence
    showProcessingMessages: false,
    messageStyle: "none",
    showMathMenu: false,
    showMathMenuMSIE: false,

    styles: {
        ".MathJax_Display": {
          clear: "both"
        }
    }

});

MathJax.Ajax.loadComplete("[MathJax]/config/local/local.js");
```

Premièrement, le groupe d'options commenté `Apparence` sert à supprimer les messages inutiles de MathJax, c'est-à-dire les messages de chargements par exemple. On supprime également le menu MathJax que celui-ci ajoute aux expressions qu'il transforme et que l'on affiche avec un clic droit. Ce menu est jugé inutile et surchargeant. Ensuite le dictionnaire `styles` permet d'ajouter du CSS en plus de celui que génère MathJax. On ajoute à la classe `MathJax_Display`, une classe utilisée sur toutes les expressions qu'il transforme, le style `clear: "both"`. Il s'agit d'une correction d'un bug du plugin, car par défaut MathJax pouvait interférer avec le style déjà en place sur le site web. Ce problème créait des problèmes gênants pour l'interface. Finalement, la dernière configuration faite n'est pas la plus simple. Nous avons vu que MathJax analysait automatiquement toute la page HTML pour transformer le contenu balisé. Nous avons modifié ce comportement par défaut de MathJax. Il s'agit d'une part d'une question de performance. En effet, nous n'avons pas envie que des pages qui n'ont pas de contenu mathématique soient analysées par le lourd code JavaScript de MathJax. D'autre part, pour des questions de contrôle, l'on veut savoir exactement où il agit. L'option `skipStartupTypeset` permet donc de désactiver l'exécution automatique du processus MathJax. Le prochain défi est de trouver un moyen facilement utilisable pour indiquer quelle partie du code HTML doit être analysée. AngularJS a un outil très pratique pour accomplir ce genre de tâches. Nous avons déjà eu l'occasion de le découvrir dans le premier chapitre : *les directives*. Ce sont des attributs ou éléments HTML qui exécutent des actions spécifiques, comme `ng-repeat` ou `ng-show`.

Nous allons donc créer une directive `mathjax`. Le code ci-dessous déclare simplement la directive.

```
// on déclare la directive
app.directive('mathjax', function() {
    // Le code vient ici
}]);
```

Le but de cette directive est de pouvoir baliser les parties de notre page HTML qui contiennent des mathématiques. Par exemple, lorsque l'on affiche un cours, on aimerait faire de la façon suivante : `<mathjax>{{ cours.contenu }}</mathjax>`. Pour définir la directive et son comportement, il faut retourner un objet JavaScript contenant les options de notre directive. Ci-dessous se trouve la directive complète que nous allons analyser.

```
1  app.directive('mathjax', function($timeout) {
2      restrict: 'AE',
3      template: '<div class="ng-hide" ng-transclude></div>',
4      transclude: true,
5      link: function(scope, element, attrs) {
6          $timeout(function () {
7              MathJax.Hub.Queue(["Typeset", MathJax.Hub, element[0]]);
8              MathJax.Hub.Queue(function() {
9                  element.children().removeClass("ng-hide");
10             });
11         });
12     }
13 });
```

— **restrict** : 'AE' signifie que notre directive peut être un attribut (A = Attribut), avec le forme `<directive></directive>` ou un élément (E = Element), avec la forme `<div directive></div>`. On peut aussi ajouter l'option C pour utiliser la directive en tant que classe (C = Class), avec la forme `<div class="directive"></div>`.
— **template** : Le contenu HTML dans notre directive.
— **tranclude** : Cette option permet de récupérer le contenu qui est dans la directive et de le réinjecter dans le template. En fait, par défaut, qu'importe le contenu de la directive sur la page HTML, celui-ci est de toute façon remplacé par la chaine de caractère de l'option `template`. Par exemple, si on écrit `<mathjax>{{ cours.contenu }}</mathjax>`, le contenu du cours est supprimé. Par contre, quand on utilise l'option `transclude`, Angular récupère le contenu de la directive et l'injecte dans notre template, à l'endroit où l'on spécifie `ng-tranclude`. Ainsi on trouve dans l'option `template` le code HTML `<div ng-transclude></div>`. Le contenu de la directive est donc ajouté dans la `div`.
— **link** : c'est la fonction qui est exécutée une fois que la page est compilée. En clair, quand AngularJS a transformé la page HTML et ses directives, le DOM est totalement généré. Ensuite seulement JavaScript peut agir sur celui-ci. Par conséquent, avec la fonction que l'on passe à `link`, on peut manipuler le contenu de notre directive. La fonction prend trois arguments. `scope` est basiquement l'objet qui contient les données du modèle. `element` est l'élément HTML lui-même et `attrs` contient les attributs HTML supplémentaires de notre directive.

Il y a encore beaucoup d'autres options disponibles pour personnaliser une directive, elles sont listées sur la documentation [3].

Intéressons nous maintenant au code qui se trouve à l'intérieur de la fonction `link`. La difficulté se trouve surtout dans le code spécifique à MathJax, `MathJax.Hub.Queue`. En fait, cette expression permet d'exécuter des fonctions en lien avec MathJax au bon moment. Elle permet tout simplement d'assurer que les fonctions que l'on passe à `Queue` s'exécutent une fois que MathJax est complétement chargé et qu'il est prêt à être utilisé. La première expression MathJax, à la ligne 7, indique qu'il faut analyser et mettre en forme le contenu de l'élément qu'on passe en argument, dans notre cas `element[0]`. On peut remarquer que l'on utilise `element[0]` et pas `element`. `element` est un objet contenant plusieurs informations tandis que `element[0]` retourne l'élément du DOM. Ensuite dans la deuxième expression, ligne 8, on enlève simplement la classe `ng-hide` de notre élément. Par défaut on cache le contenu de la directive, comme on peut l'observer dans l'option `template` qui contient la classe `ng-hide`. Cette expression sert à afficher la directive seulement une fois que les expressions mathématiques ont été transformées. Ainsi, l'utilisateur ne voit pas du contenu qui n'a pas encore été formaté par MathJax. Pour plus d'informations sur la `Queue` MathJax, on peut se rendre sur la documentation officielle [4]. Finalement, le code est enveloppé dans une fonction `$timeout` qui permet simplement d'assurer, lorsque notre directive est utilisée dans une boucle, que la boucle soit terminée avant que nous exécutions les transformations.

Notre directive est prête à être utilisée ! Maintenant, il suffit de l'utiliser pour mettre en forme notre contenu mathématique à l'endroit où on le désire.

```
<body>
    <mathjax>
        $$ f(x) = 5x + 4 $$
    </mathjax>
</body>
```

3. https://docs.angularjs.org/guide/directive. Consulté le 28 décembre 14.
4. http://docs.mathjax.org/en/latest/typeset.html. Consulté le 28 décembre 14.

CHAPITRE 9

Tests

Une série de tests a été écrite afin de pouvoir garantir l'utilisation des fonctionnalités et faciliter les futurs apports au projet. Dans notre cas, il s'agit de test système, *end to end* en anglais. Ce type de tests signifie que l'on teste l'application dans son ensemble et d'un point de vue utilisateur. On ne va pas tester juste une partie du code comme dans des tests unitaires, mais l'on va simuler un utilisateur et naviguer sur notre site afin de s'assurer que les fonctionnalités sont opérationnelles.

AngularJS fournit un outil adapté pour tester ses applications qui se nomme Protractor[1]. Ce programme nous permet d'écrire des tests adaptés à AngularJS. En fait, Protractor fournit des outils spécifiques au framework. Par exemple, il permet de sélectionner des éléments par leur modèle Angular, `ng-model="password"`. Protractor attend aussi automatiquement qu'Angular ait fini de préparer la page et que les requêtes AJAX soient terminées avant de faire les tests. Basiquement, un test consiste tout simplement à se rendre sur une page, à cliquer sur un bouton et à vérifier un résultat. On simule un utilisateur. Dans le projet, les tests se trouvent dans le dossier `courses/spec`. Le fichier `conf.js` sert à configurer Protractor et les autres fichiers se terminant par *_spec.js* contiennent les tests. Chaque fichier correspond à une route de notre site. Prenons un exemple dans lequel on vérifie si l'utilisateur peut changer de page sur un cours.

```
it("allows to switch pages", function() {
    // On se rend sur la page
    browser.get("http://localhost:3333/courses/#/1/view/1");
    // On clique sur le lien
    element(by.id("next-page")).click();
    // On s'attend à ce que le titre ait changé
    expect(element.all(by.binding("page.name")).getText()).toEqual("Les équations");
});
```

Notre test commence par la fonction `it`, qui décrit ce que l'on teste, par exemple dans ce code on déclare que "Cela permet de changer de pages". Cette description est utile car elle nous permet de savoir rapidement ce qui est testé. Aussi, lorsqu'un test ne fonctionne pas, de savoir tout de suite quelle fonctionnalité ne marche plus. Ensuite dans la fonction *it* on décrit une série d'étape. On se rend sur la page `courses/#/1/view/1`, on trouve le lien avec l'id `next-page` et on clique dessus. A la fin du test, on écrit une "attente", c'est ce qui détermine si le test passe. Dans notre cas, l'on s'attend à ce que le nouveau titre de la page soit "Les équations", car on a changé de page en cliquant sur le bouton.

Pour lancer les tests écrits pour notre application, on lance la commande *python3 manage.py tests*. La commande se charge principalement de créer une base de données propre avec des données spécifiques et de lancer les serveurs. Une fois la commande lancée, une fenêtre de navigateur s'ouvrira et les tests défileront dedans comme si un utilisateur agissait. A la fin des tests, il suffit de revenir dans la console pour avoir le compte-rendu, les résultats.

La commande exécutera par défaut tous les tests se trouvant dans le dossier `courses/spec`. Pour changer ce comportement, on peut spécifier le fichier en argument : `python3 manage.py tests home_spec.js`. Pour changer le dossier, par défaut `courses`, il faut le spécifier avec l'option `--app`. Par exemple : `python3 manage.py tests --app teachers`.

1. http://angular.github.io/protractor/#/. Consulté le 18 mars 15.

CHAPITRE 10

Guide du développeur

10.1 Fichiers

Cette section permet de s'y retrouver dans la multitude de fichiers du projet et de comprendre certains processus.

- **courses/**
  - **admin.py** : permet de rajouter les modèles que l'on veut voir apparaitre et modifier dans la zone d'administration Django. Documentation officielle [1]
  - **api.py** : regroupe toutes les vues génériques RestLess qui servent à construire l'API JSON nécessaire à l'application AngularJS. Les classes déclarées sont ensuite utilisées dans le fichier `urls.py`.
  - **forms.py** : déclare les formulaires Django nécessaires pour enregistrer les données d'une requête dans la base de données. Ils sont utilisés principalement dans le fichier *api.py*. Documentation officielle [2]
  - **models.py** : déclare les modèles de notre application. Contient également des méthodes d'instance pour certains modèles. Documentation officielle [3]
  - **urls.py** : contient les URL spécifiques à l'application. Elles sont ensuite ajoutées dans le fichier principal `webmath/urls.py`. La première url, de nom `index`, est le point de départ de notre application AngularJS. Le reste des routes est défini directement par AngularJS dans le fichier `courses/static/courses/javascripts/config/routes.js`. Dans `urls.py`, la deuxième url, de nom `pdf`, est celle qui génère le PDF d'un cours. Les routes qui suivent sont celles de l'API JSON qui utilisent les vues génériques du fichier `api.py`. Documentation officielle [4]
  - **utils.py** : regroupe une série de fonctions utiles utilisées à travers l'application.
  - **views.py** : contient les vues Django. A l'instar des URL, il n'y a que deux fonctions, une qui est le point de départ de l'application et l'autre qui génère le PDF d'un cours. Documentation officielle [5]
- **courses/templates/courses/** : contient le gabarit de base `courses.html` de notre application et le fichier Markdown `pdf.md` servant à générer le PDF d'un cours.
- **courses/static/courses/**
  - **html/** : contient tous les fichiers HTML utilisés par AngularJS. Les pages HTML sont reliées à une route dans le ficher `courses/static/courses/javascripts/config/routes.js` d'AngularJS qui s'occupe d'associer une route à un fichier HTML.
  - **images/** : contient les images utilisées pour le design de l'application.
  - **stylesheets/** : contient les feuilles de styles.
- **courses/static/courses/javascripts/ :**
  - **config/routes.js** : déclare les routes principales de notre application avec AngularJS. Documentation officielle [6]
  - **controllers/** : déclare les contrôleurs AngularJS. Un fichier correspond à une route et son contrôleur. Ils sont utilisés dans le fichier

1. https://docs.djangoproject.com/fr/1.7/ref/contrib/admin. Consulté le 14 mars 15.
2. https://docs.djangoproject.com/fr/1.7/topics/forms/. Consulté le 14 mars 15.
3. https://docs.djangoproject.com/fr/1.7/topics/db/models/. Consulté le 14 mars 15.
4. https://docs.djangoproject.com/fr/1.7/topics/http/urls/. Consulté le 14 mars 15.
5. https://docs.djangoproject.com/fr/1.7/topics/http/views/. Consulté le 14 mars 15.
6. https://docs.angularjs.org/tutorial/step_07. Consulté le 14 mars 15.

`courses/static/courses/javascripts/config/routes.js`.
    - **directives/** : déclare des directives AngularJS. Documentation officielle [7]
    - **factories/resources.js** : déclare des objets ressources qui permettent de communiquer facilement avec l'API. Documentation officielle [8]
    - **filters/** : déclare des filtres AngularJS. Documentation officielle [9]
    - **app.js** : point de départ, déclare l'application AngularJS principale ainsi que les modules avec leurs dépendances. C'est dans ces modules qu'on ajoute ensuite les filtres, contrôleurs, etc.
    - **extensions.js** : permet de personnaliser Showdown.js, la bibliothèque qui transforme le Markdown en HTML. Documentation officielle [10]
- **spec/conf.js** : fichier de configuration de Protractor, le moteur de test.
- **spec/** : contient les fichiers de tests end-to-end Protractor. Chaque fichier correspond à une route de l'application.
- **management/commands/**
    - Documentation officielle [11]
    - **seed.py** : commande qui crée des données de démonstration dans la base de données afin d'avoir une application fonctionnelle.
    - **tests.py** : commande qui lance les tests Protractor.
- **webmath/test_router.py** : Le routeur permet d'utiliser plusieurs bases de données avec Django. En l'occurrence, le routeur permet d'utiliser une base de données différente lorsqu'on lance les tests de notre application. Documentation officielle [12]

10.2 URL

Cette section regroupe une explication de toutes les URL de l'application. Toutes les URL se trouvent dans l'espace de nom `courses`, par exemple `/courses/help`. Lorsque un `#` se trouve dans l'URL, cela signifie que c'est une URL définie par AngularJS.

10.2.1 Utilisateurs

- **#/** : Page d'accueil de l'application, on y trouve une liste des cours publiés. Ils peuvent être triés par catégories ou favoris.
- **#/new** : Page qui permet aux enseignants de créer un nouveau cours.
- **#/ :course_id/edit/ :page** : Page d'édition d'un cours. L'enseignant peut y éditer le contenu de son cours, le publier ou le retirer.
- **#/ :course_id/preview/ :page** : Prévisualise une page d'un cours. Lorsqu'un enseignant rédige un cours, il peut voir le résultat final grâce à cette page.
- **#/teacher/courses :** Liste de tous les cours du site, publiés ou non.
- **#/help** : Page qui fournit une aide aux rédacteurs concernant la syntaxe *Markdown* et *LaTex*.
- **#/ :course_id/edit** : Page qui permet de modifier les informations de base d'un cours, telles que le nom, la description ou la difficulté.
- **#/ :course_id/show/ :page** : Page pour lire un cours. Le cours y est affiché, on peut naviguer à travers les pages, commenter le cours, etc.
- **/pdf/ :course_id/*.pdf** : Renvoie un cours au format PDF pour pouvoir être téléchargé.
- **#/about** : Page d'information générale sur l'application.

10.2.2 API

Les URLs de l'API sont dans l'espace de nom `api`, par exemple `/courses/api/themes`.
- **/courses/all** : renvoie une liste de tous les cours, publiés ou non.

7. https://docs.angularjs.org/guide/directive. Consulté le 14 mars 15.
8. https://docs.angularjs.org/api/ngResource/service/$resource. Consulté le 14 mars 15.
9. https://docs.angularjs.org/guide/filter. Consulté le 14 mars 15.
10. https://github.com/showdownjs/showdown. Consulté le 14 mars 15.
11. https://docs.djangoproject.com/fr/1.7/howto/custom-management-commands/. Consulté le 14 mars 15.
12. https://docs.djangoproject.com/fr/1.7/topics/db/multi-db/. Consulté le 15 mars 15.

- **/courses**
  - GET : renvoie tous les cours publiés.
  - POST : crée un nouveau cours.
- **/courses/ :id**
  - PUT : met à jour un cours.
- **/pages/ :page_id/courses/ :course_id** :
  - GET : renvoie le contenu d'une page d'un cours.
  - PUT : met à jour le contenu d'une page.
- **/themes** : renvoie une liste de tous les thèmes en incluant leurs chapitres respectifs.
- **/pages/ :page_id/sections**
  - POST : ajoute une section à une page d'un cours.
- **/courses/ :course_id/pages**
  - POST : ajoute une page à un cours.
- **/sections/ :id**
  - DELETE : supprime une section.
- **/courses/ :course_id/comments**
  - GET : renvoie les commentaires d'un cours.
  - POST : ajoute un commentaire à un cours.
- **/courses/ :course_id/menu** : Permet de construire le menu d'un cours en renvoyant le nom de ses pages et de leurs sections.
- **/courses/ :course_id/publish**
  - PUT : publie/retire un cours en changeant l'attribut `published` de `True` à `False` et vice-versa.
- **/courses/ :course_id/favorite**
  - PUT : ajoute/retire un cours au/des favoris de l'utilisateur.
- **/pages/ :page_id/progression**
  - POST : marque une page d'un cours comme comprise ou à relire pour l'utilisateur.

10.3 Concepts

10.3.1 Intégration d'AngularJS avec Django

En dehors de l'API et des PDF, Django ne fournit qu'une seule route dans l'application. En effet, à partir de cette route, Angular s'occupe de gérer les autres routes et les templates. Concrètement, lorsqu'on charge une page de notre application, la requête va d'abord passer par la vue Django `index` déclarée dans le fichier *views.py*. Cette vue s'occupe simplement d'afficher le template `courses.html`. Ce fichier HTML est un layout pour notre application, c'est-à-dire que son contenu est sur toutes les pages. Il contient le menu, l'inclusion des fichiers JavaScript et des feuilles de syle, ainsi que le pied de page. Dans la balise `body`, on a ajouté la directive Angular `ng-app=Courses`. On déclare qu'à l'intérieur de cette balise se trouve une application AngularJS nommée `CoursesApp`. Ainsi, une fois que Django a affiché le template `courses.html`, Angular va insérer le contenu du bon fichier HTML dans la balise `body` selon l'URL et les routes écrites dans le fichier `routes.js`. La page finale est maintenant visible par l'utilisateur. Par exemple, si l'on se rend sur `courses/help`, Angular s'occupe de chercher le fichier `help.html` et d'insérer son contenu dans la balise `body` de `courses.html`. L'avantage de ce système est que lorsqu'on change de page, la vue Django n'est pas rappelée, mais seul le contenu de *body* est mis à jour avec le contenu HTML approprié à l'URL. AngularJS rend ainsi notre site web plus rapide.

CHAPITRE 11

Bogues et améliorations

11.1 Bogues

Bogues éventuels dans l'application.

— L'intégration de la syntaxe Markdown n'a pas été totalement testée, il est possible que certaines formules ne marchent pas lors de l'affichage du cours.
— La génération de PDF n'a également pas été complétement testée, il est également possible de trouver des bugs, comme avec les images ou les mathématiques contenus dans les cours.

11.2 Améliorations

— Implémentation complète de l'authentification
— Possibilité de changer l'ordre des pages
— Possibilité d'uploader des vidéos et des images
— Proposer des exercices en lien avec le cours
— Afficher des statistiques (progression, pages à relire, propositions...) dans le Dashboard élève
— Possibilité pour les étudiants de demander des cours spécifiques
— Possibilité de faire une recherche parmi les cours publiés

CHAPITRE 12

Conclusion

CHAPITRE 13

Bibliographie

13.1 Bibliographie

GREEN Brad & SESHADRI Shyam, AngularJS, Sebastopol, O'Reilly Media, 2013, 183 p.

ALEXIS Pierre & BERSINI Hugues, Apprendre la programmation web avec Python et Django, Paris, Eyrolles, 2012, 319 p.

LE GOFF Vincent, Apprenez à programmer en Python, Paris, OpenClassrooms, 2012, 382 p.

13.2 Webographie

Python : https://docs.python.org/3/

Django : https://www.djangoproject.com/

AngularJS : https://angularjs.org/

Protracor : http://angular.github.io/protractor/#/

CSS & HTML : http://www.w3schools.com/

Bootstrap : http://getbootstrap.com/

Restless : https://django-restless.readthedocs.org/en/latest/

Pandoc : http://johnmacfarlane.net/pandoc/

Showdown : https://github.com/showdownjs/showdown

MathJax : http://www.mathjax.org/

Git : http://git-scm.com/

CHAPITRE 14

Source des illustrations

Tables

3.1 One-way data binding . . . 9
3.2 Two-way data binding . . . 10

4.1 Création d'un cours . . . 14
4.2 Edition d'un cours . . . 15
4.3 Tous les cours des professeurs . . . 15
4.4 Tous les cours . . . 16
4.5 Lire un cours . . . 17
4.6 Les commentaires . . . 17

5.1 Cas d'utilisation . . . 20
5.2 Schéma de navigation . . . 21

6.1 Schéma résumant une base de données relationnelle . . . 23
6.2 Schéma de toutes les tables du modèle relationnel . . . 24
6.3 Schéma qui résume les relations des tables courses, pages et sections . . . 25

7.1 Schéma de la communication entre un client et un serveur . . . 31
7.2 API : schéma du traitement d'une requête à l'aide des vues génériques RestLess . . . 32
7.3 Surcharge d'une méthode . . . 33

8.1 Résultat de MathJax . . . 35