Développement d'une application de quiz

Travail de maturité

Benoît Léo Maillard

27 mars 2015

Table des matières

CHAPITRE 1

Présentation de RapydScript

1.1 Pourquoi utiliser Python plutôt que Javascript ?

Les sites web d'aujourd'hui utilisent de plus en plus le javascript afin de rendre la navigation plus confortable pour le visiteur. Les fonctionnalités offertes par ce language permettent de concevoir une grande variété d'applications utilisables directement dans le navigateur, qui s'éxécutent côté client et qui sont donc par conséquent très accessibles au grand public. Le Javascript est donc un langage quasiment incontournable pour toute personne qui s'intéresse au développement web.

Malgré cela, ce langage n'est pas forcément facile à aborder pour un programmeur habituée à coder en Java, en Python ou dans un autre langage de programmation moderne, car son fonctionnement diffère dans un certain nombre d'aspects fondamentaux. Par exemple, le Javascript est un langage orienté objet à prototype, c'est à dire que les objets utilisés en Javascript sont des copies d'objets prototype, et non des instances de classes comme en Python et Java. On peut également mentionner d'autres différences importantes, comme la portée des variables par défaut. Afin de palier aux difficultés que peut rencontrer un développeur qui débute dans ce langage, divers outils sont apparus pour tenter de remplacer Javascript par Python, avec pour objectif de combiner les possibilités offertes par le Javascript avec la simplicité et la clarté de Python. Ce travail a pour but de présenter et expliquer les fonctionnalités de RapydScript, un outil permettant d'écrire du code très semblable à du code Python (dans sa syntaxe et sa philosophie) et de générer du Javascript à partir de ce code.

1.2 Différentes manières d'aborder le problème : CoffeeScript, Brython et RapydScript

Un certain nombre d'outils open-source tentent de faciliter l'écriture de code Javascript et différentes approches du problème sont possibles. CoffeeScript (http://coffeescript.org/) est décrit par ses auteurs comme un langage à part entière, avec sa propre syntaxe (qui ressemble un peu à celle de python, mais qui s'inspire surtout de ruby), qui peut être compilé en Javascript. CoffeScript permet l'écriture d'un code source plus clair et concis, le code généré est éxecuté avec les mêmes performances que du code javascript natif (puisqu'il s'agit simplement de javascript), mais son utilisation nécessite l'apprentissage d'un nouveau langage, ce qui peut être un problème pour un débutant.

À l'opposé, Brython (http://www.brython.info/) propose une toute autre approche : le développeur peut écrire du code en Python qui sera éxecuté par un interpréteur Python entièrement intégré dans la page web codé en javascript. C'est certainement la solution la plus fidèle à Python, puisque la syntaxe de Brython est exactement identique à celle de Python. L'accès au DOM et l'utilisation d'Ajax se fait via l'import de modules externes. Brython est donc idéal pour quelqu'un qui ne connaît pas Javascript mais possède de bonnes bases en Python. Cependant, comme la traduction en Javascript se fait "en live", l'impact sur les performances se fait ressentir (http://pyppet.blogspot.ch/2013/11/brython-vs-pythonjs.html) et peut poser problèmes pour des scripts complexes. Le script de l'interpréteur (environ 10'000

lignes) doit également être inclus dans chaque page html qui comporte du code Brython. De plus, il n'est pas possible d'utiliser des libraires Javascript externes, telles que jQuery ou AngularJS.

RapydScript a l'avantage de combiner les qualités des deux outils cités plus haut, en permettant d'écrire du code très similaire à du code Python standard et en le compilant en Javascript. Rapydscript est donc facile à prendre en main pour n'importe quelle personne sachant coder en Python et ne limite pas la rapidité d'éxecution puisque le code qui sera executé est tout simplement du Javascript. Il est aussi possible d'appeler n'importe quelle fonction Javascript standard et par extension d'utiliser n'importe quelle librairie externe. Ce dernier point n'est pas négligeable puisque j'ai opté pour jQuery pour développer mon application de création de quiz en ligne. Le choix de RapydScript pour la réalisation de ce travail est apparu comme évident après avoir considéré les qualités et défauts de ces différents outils. La prise en main ainsi que la compréhension de son fonctionnement a pu se faire aisément, d'autant plus que l'absence de documentation ou de tutoriel complet sur cette technologie en français constitue une motivation supplémentaire et donne un aspect inédit à ce travail.

1.3 Introduction à RapydScript

1.3.1 Installation

1.3.2 Compilation

1.3.3 Programmer avec RapydScript

Notions de base

Pour un habitué de Python, la prise en main de RapydScript peut se faire très rapidement : il suffit d'écrire son programme comme si on écrivait un programme Python, même si dans certains cas particuliers il n'est pas possible de faire exactement la même chose avec RapydScript. Ces cas particuliers seront abordés plus loin.

Pour commencer avec un exemple simple, voici en python une fonction qui prend nombres en argument et retourne le plus grand. La fonction est ensuite appelée. Ce code est parfaitement valide en Python.

```
def maximum(n1, n2):
    if n1 >= n2:
        return n1
    elif n2 > n1:
        return n2

maximum(n1, n2) #Appel de la fonction
```

Une fois la compilation effectuée avec RapydScript, on obtient ce résultat :

```
function maximum(n1, n2) {
    if (n1 >= n2) {
        return n1;
    } else if (n2 > n1) {
        return n2;
    }
}
maximum(5, 18);
```

On peut voir les opérations qu'a fait RapydScript "traduire" le code source en Javascript : Remplacer le mot clé `def` par `function`, ajouter les accolades qui englobent la fonction et les structures `if`, ajouter des parenthèses autour des conditions et ajouter un `;` à la fin de chaque instruction. On remarque aussi que les commentaires ont été supprimés,

puisqu'ils sont seulement utiles dans le code source. Le code ainsi produit peut maintenant être éxecuté dans n'importe quel navigateur.

Cet exemple montre cependant un cas assez peu significatif de la puissance de RapydScript puisque le code Javascript correspondant au code source est quasiment identique. Mais RapydScript permet aussi de compiler du code typiqement Python, comme par exemple des boucles for, qui n'ont pas d'équivalent en javascript.

```
names_list = ["Paul", "Marie", "Pierre", "Lucie"]

for name in names_list:
    print(name)
```

RapydScript produit un code équivalent en Javascript qui s'éxecutera comme en Python. Le code généré est cette fois plus complexe et des connaissances en Javascript sont nécessaires pour le comprendre. Ce type de manipulations sera étudié dans un chapitre ultérieur. On peut par exemple aussi implémenter une fonction avec des arguments qui prennent des valeurs par défaut, ce qui n'est pas possible en Javascript.

Programmation orientée objet (POO)

Ce qui fait de RapydScript un outil si puissant est principalement les possibilités qu'il offre pour faire de la POO. En Javascript, la POO est basée sur le prototypage, et il est beaucoup plus complexe de créer ses propres objets, avec de l'héritage, etc. En python, cela est beaucoup plus simple, il est donc particulièrement intéressant de pouvoir utiliser la POO Python pour faire de la programmation web front-end.

Encore une fois, il suffit d'écrire une classe comme on le ferait en Python :

```
class MyObject:
    def __init__(self, name):
        self.name = name

    def get_name(self):
        return self.name

object = MyObject("Object 1") #Instanciation avec un paramètre
object.get_name() #Retourne "Objet 1"
```

Il est également possible de faire de l'héritage :

```
class MyObjectPlus(MyObject): #Classe qui hérite de la classe créée précédemment
    def __init__(self, name, number):
        MyObject.__init__(self, name)
        self.number = number

    def get_number(self):
        return self.number

    def informations(self):
        return "Nom : " + self.name + " /// Nombre : " + str(self.number)

objet = MyObjectPlus("Objet 2", 5)
objet.get_name() #Retourne "Objet 2"
objet.get_number() #Retourne 5
objet.informations() #Retourne "Nom : Objet 2 /// Nombre : 5"
```

Il n'est par contre pas possible de définir des variables de classes avec RapydScript (si on définit une variable de classe, RapydScript l'ignore simplement). Cela est dû au fait qu'en Javascript, un objet n'est pas une instance d'une classe comme en Python, mais une copie d'un objet prototype. En Python, une variable de classe est en fait un attribut de

l'objet `Class`. Il n'y a donc qu'une seul espace en mémoire pour cette variable. En Javascript, tous les attributs du prototype sont copiées à chaque fois qu'un objet est créé et il n'y a aucun équivalent aux variables de classe.

Utilisation de la bibliothèque standard Python

Avec RapydScript, il est possible d'utiliser dans le code des fonctions provenant de différentes sources. La première est la bibilothèque standard de Python, c'est à dire les fonctions natives Python, telles que `print()`, `len()` ou `range()`.

Pour utiliser les fonctions de la bibliothèque standard Python, il faut placer l'instruction suivante au début du fichier Python :

```
import stdlib
```

RapydScript définira ainsi ces fonctions dans le fichier généré et leur comportement sera en quelque sorte simulé en Javascript. Ces fonctions peuvent donc être utilisés comme on le ferait en Python, dans le code source.

Séparer son code en plusieurs fichiers

Il est déconseillé d'écrire tout son code dans un seul fichier lorsqu'on travaille sur un gros projet. Pour séparer son code en plusieurs fichiers, RapydScript prévoit un système d'imports qui ressemble à celui de Python. Voici comment procéder pour écrire son code dans plusieurs fichiers.

Premièrement, chaque fichier du code source doit utiliser l'extension *.pyj*, qui est l'extension des fichiers RapydScript. Ensuite, ces fichiers peuvent être utilisés comme des modules et être importés depuis un autre fichier. Par exemple, si on a placé une partie de notre code dans un fichier *moduletest.pyj*, on ajoutera l'instruction suivante en début de fichier :

```
import moduletest
```

Lors de la compilation, RapydScript va rassembler tous les fichiers du code source dans un même fichier Javascript, ce qui facilite aussi l'insertion du script dans un fichier HTML. Il est important de noter, cependant, que l'import d'un module ne rend pas ses fonctions disponibles dans un namespace distinct (ce qui est le cas en Python par contre). Par exemple, pour appeler la fonction `test()` du module de l'exemple précédent, voici comment procéder :

```
import moduletest

test() #Correct
moduletest.test() #Ne fonctionne pas
```

Utilisation de fonctions Javascript natives ou provenant de librairies externes

Il est également possible d'utiliser des fonctions Javascript natives, par exemple :

```
#Ces deux expressions sont équivalentes :
console.log("Bonjour") #Fonction javascript
print("Bonjour") #Fonction python
```

L'exemple parle de lui-même et ne nécessite pas d'explications supplémentaires. Il peut parfois être pratique d'utiliser des fonctions qui n'ont pas d'équivalent en Python.

Mais une autre grande force de RapydScript est la possibilité d'utiliser des librairies Javascript externes, telles que jQuery ou AngularJS. Pour cela, rien de plus simple, il suffit d'insérer le script (par exemple jQuery) que l'on veut utiliser dans le code HTML, comme ceci :

```
<html>
<head>
    <script src="jquery.js"></script><!-- jQuery -->
    <script src="myscript.js"></script><!-- Script créé avec RapydScript -->
</head>
<body>
    <div id="mydiv"></div>
</body>
</html>
```

On peut maintenant utiliser les fonctions jQuery dans notre code. Cette fonction sélectionne le et y insère du texte :

```
def add_text(text):
    $("#mydiv").text(text)

add_text("Hello World")
```

On peut procéder de la même manière pour n'importe quelle autre librairie externe Javascript.

Débugger avec RapydScript

Cas problématiques

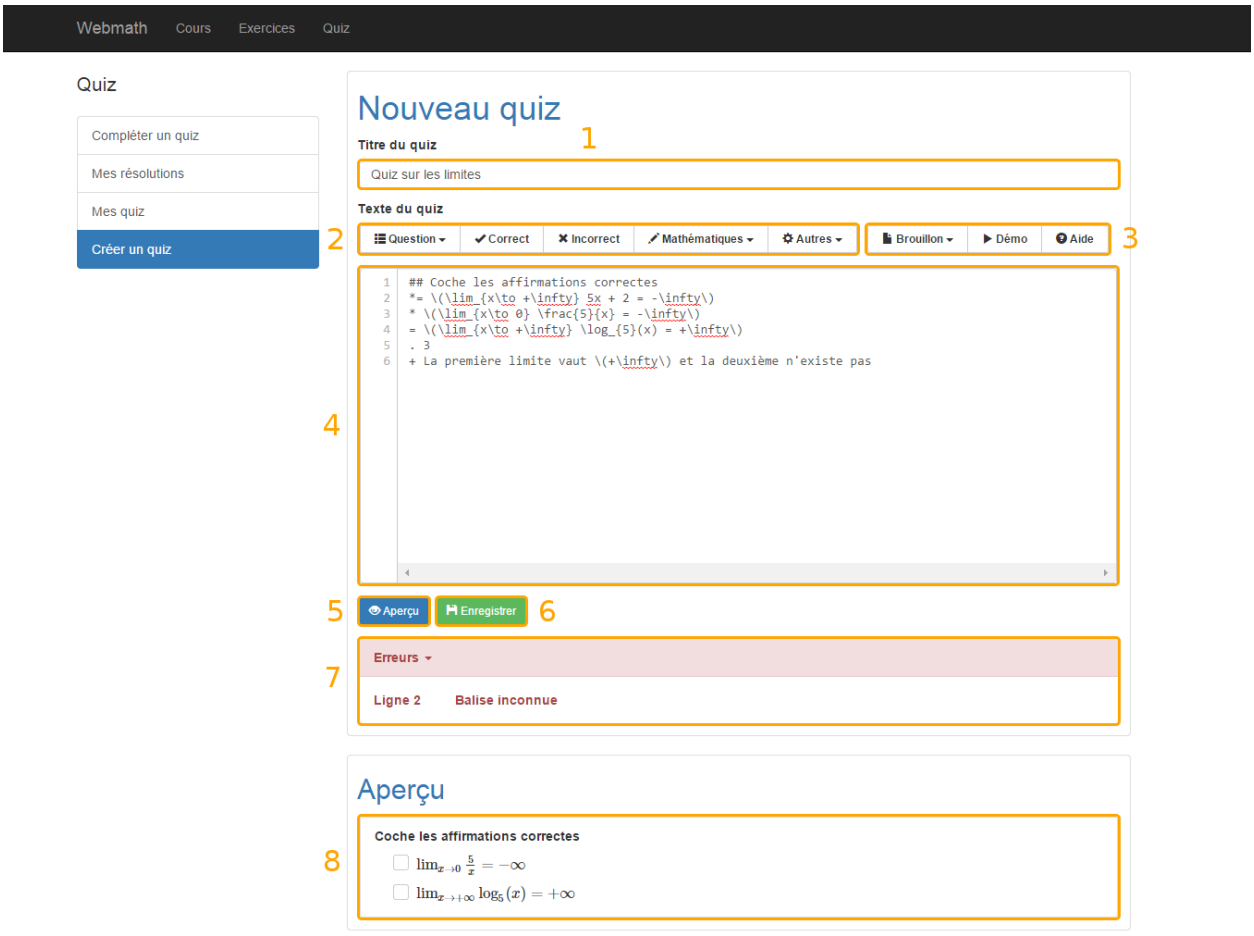
1.4 Comparaison du code source et du code généré

CHAPITRE 2

Documentation utilisateur

2.1 Professeurs

2.1.1 Création de quiz



Création d'une question et définition de ses caractéristiques

La création du quiz se fait par l'intermédiaire d'un langage de balisage de type Markdown pour définir les différentes questions du quiz ainsi que leurs caractéristiques. La définition des questions se fait selon le schéma suivant : la première ligne du paragraphe sert à indiquer le type de question et l'énoncé, alors que les lignes suivantes décrivent les solutions ou options possibles ainsi que d'autres paramètres, comme le nombre de points attribués pour la question. Un double retour à la ligne marque le passage à la question suivante.

Au début de chaque ligne, une balise suivie d'un espace indique la fonction de la ligne en question.

Explication des balises

Balise	Signification
##	Question à choix multiple avec plusieurs options qui peuvent être choisies
**	Question à choix multiple avec une seule option qui peut être choisie
??	Question avec un champ de texte à remplir
*	Option invalide dans un QCM
=	Option valide dans un QCM
=	Réponse correcte dans une question à champ de texte
.	Permet de définir le nombre de points sur la question (par défaut, 1)
+	Ajout d'un commentaire d'explication qui sera affiché lors de la correction

Exemple de quiz avec le système de balisage

```
## Énoncé de la question à choix multiple (plusieurs cases peuvent être cochées)
* Option 1
= Option 2 (correcte)
= Option 4 (correcte)
+ Commentaire affiché à la correction
. 1.5

** Énoncé de la question à choix multiple (une seule case peut être cochée)
* Option 1
* Option 2
= Option 3 (correcte)
. 2

?? Question à réponse courte
= Réponse correcte
= Autre réponse correcte possible
```

Pour la première question, la balise `##` indique qu'il s'agit d'une question à choix multiple avec plusieurs réponses possibles, alors qu'une option incorrecte et deux options correctes sont respectivement définies avec les balises * et =. Ensuite, on a ajouté un commentaire avec + et défini le nombre de points avec le symbole `.`. Il est important de retenir que toutes les balises sont suivies d'un espace, sans quoi elles ne sont pas reconnues.

La barre d'outils située en dessus de la zone de texte offre la possibilité de créer un quiz sans maîtriser le système de balisage. Pour ajouter une question, il suffit de cliquer sur l'onglet *Question* et de choisir le type de question souhaité dans le menu déroulant. La balise est insérée automatiquement et il n'y a plus qu'à écrire l'énoncé de la question. Il en va de même pour ajouter des solutions ou des options avec les boutons *Correct* et *Incorrect*. Les retours à la lignes et les espaces sont placés automatiquement avant et après la balise si cela est nécessaire. Le menu *Autres* permet de choisir le nombre de points à attribuer pour chaque question et d'écrire un commentaire destiné à être affiché lors de la correction automatique du quiz. Ce commentaire est censé apporter une justification à la solution de la question ou une aide pour les élèves n'ayant pas répondu correctement.

Types de questions

Question à réponse courte La question à réponse courte se présente sous la forme d'un simple champ de texte à compléter. La balise caractérisant ce type de questions est la balise `??`. Ensuite, une ou plusieurs réponses peuvent être définies comme correctes à l'aide de la balise =. S'il y a plusieurs solutions, elles doivent être séparées par un retour à la ligne et chacune doit être précédée de la balise =. Si la réponse donné par l'élève correspond exactement à une des solutions, il obtient tous les points. Il est conseillé de ne pas définir des solutions complexes ou trop longues pour éviter de compter comme une erreur l'absence de virgule, de point ou d'un autre caractère spécial. Les réponses apportées par les élèves définies comme incorrectes lors de la correction automatique pourront toutefois être admise en tant que solution plus tard.

IMAGE ICI

Question à choix multiples avec un seul choix valide Pour ce type, plusieurs options sont affichées et l'élève ne peut en sélectionner qu'une. La balise associé à ce type est la balise `**`. Une seule option valide doit donc être définie avec la balise =, toutes les autres doivent être erronnées et donc précédées par la balise `*`. L'élève reçoit tous les points s'il sélectionne la bonne solution, et aucun point dans tous les autres cas.

IMAGE ICI

Question à choix multiples avec un seul choix valide Définie par la balise `##`, il s'agit d'une question semblable à la précédente mais l'élève a cette fois la possibilité de choisir plusieurs options. Les options qui doivent être sélectionnées sont définies avec la balise = et les autres avec la balise `*`. Le professeur doit cependant définir au moins une option correcte. Lors de la correction, l'élève peut obtenir des points pour un choix qu'il a coché et que le professeur a défini comme correct et inversément, c'est à dire qu'il peut aussi gagner des points sur un choix qui n'est pas sélectionné, à condition qu'il soit défini comme erronné.

IMAGE ICI

Affichage de l'aperçu et des erreurs

Il est possible à tout moment d'afficher un rendu du quiz tel que le verront les étudiants en cliquant sur le bouton *Aperçu* en dessous de la zone de texte. On peut ainsi voir si toutes les questions s'affichent comme prévu et également détecter les éventuelles erreurs dans le code. Ces erreurs s'affichent dans l'encadré rouge en dessous du bouton *Aperçu*. Pour chaque erreur, un message explicatif apparaît accompagné du numéro de ligne où s'est produite l'erreur.

IMAGE EXEMPLE

Dans cet exemple, on a tenté d'utiliser la balise `*=`, qui n'existe pas. C'est pourquoi on obtient le message suivant : *Balise inconnue*.

Un quiz ne peut pas être envoyé et enregistré dans la base de données tant qu'il comporte encore des erreurs. C'est pourquoi elles doivent être impérativement corrigés pour valider l'envoi lors du cliq sur le bouton *Enregistrer*.

Affichage de mathématiques

Il est possible d'afficher des formules mathématiques à l'aide de la bibliothèque Javascript MathJax. Cet outil permet d'écrire des expressions sous forme de LaTex et de les convertir en HTML pour qu'elles soient visibles dans le navigateur. Il existe deux méthodes d'affichage proposées par MathJax : la méthode *in-line* et la méthode *displayed*. La première méthode offre la possibilité d'inclure une formule dans un paragraphe de texte. Les formules en *in-line* doivent être entourées des caractères suivants : `\(...\)`. Avec la méthode *displayed*, les expressions sont affichées en plus grand, centrées et détachées du reste du texte. Les formules utilisant cette méthode sont délimitées par les balises `$$...$$`.

La barre d'outils propose un menu dédié à l'affichage des mathématiques. Deux boutons permettent d'insérer les délimiteurs des méthodes *in-line* et *displayed* et d'autres options pour afficher un échantillon de formules et de symboles sont disponibles. Cette liste est toutefois non-exhaustive et il est conseillé de se référer à la documentation de LaTex [#latex-doc] pour obtenir des informations plus précises à ce sujet.

Voici un exemple de question comportant l'affichage de limites :

```
## Coche les affirmations correctes
* \(\lim_{x\to +\infty} 5x + 2 = -\infty\)
* \(\lim_{x\to 0} \frac{5}{x} = -\infty\)
= \(\lim_{x\to +\infty} \log_{5}(x) = +\infty\)
```

Résultat lors de l'aperçu :

Coche les affirmations correctes

- ☐ $\lim_{x \to +\infty} 5x + 2 = -\infty$
- ☐ $\lim_{x \to 0} \frac{5}{x} = -\infty$
- ☐ $\lim_{x \to +\infty} \log_5(x) = +\infty$

Enregistrement et importation de brouillons

Envoi définitif du quiz

Lorsque l'édition du quiz est terminée et que toutes les questions sont prêtes, le quiz peut être envoyé afin d'être sauvegardé dans la base de données et disponibles à la résolution pour les élèves. Avant d'envoyer un quiz, il faut s'assurer d'avoir défini un titre **(1)** et d'avoir corrigé toutes les éventuelles erreurs présentes dans le code **(7)**. Lors du cliq sur le bouton *Enregistrer* **(6)**, un avertissement apparaîtra au cas où des erreurs persistent et l'envoi ne pourra pas se faire.

2.1.2 Suivi des élèves

Liste des quiz créés par un professeur

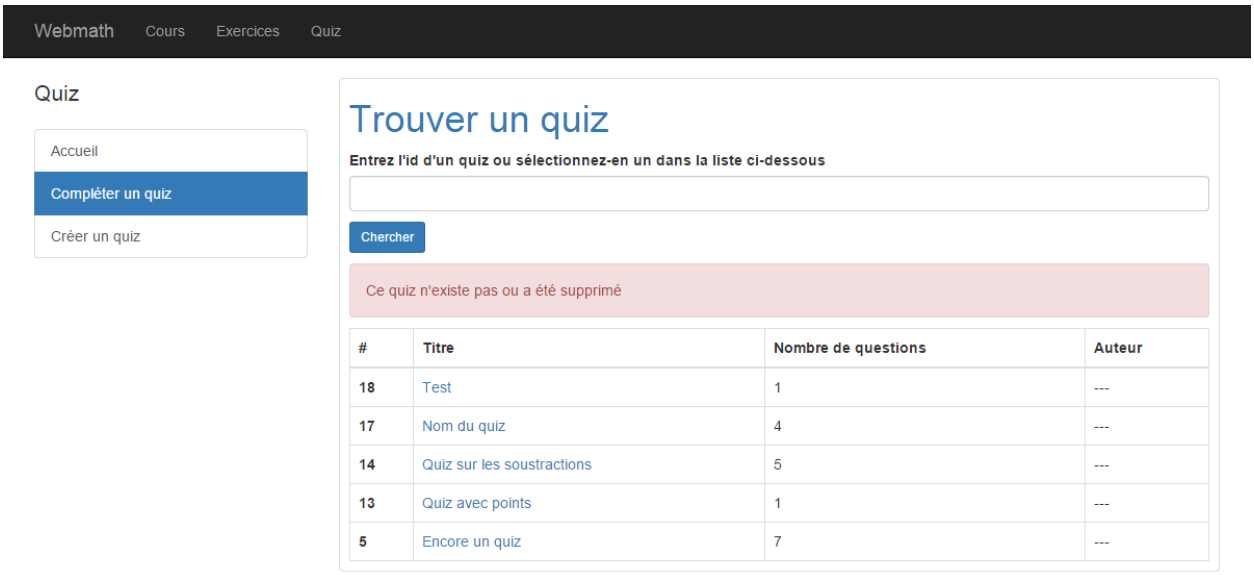
Affichage des statistiques avancées

Consultation des corrections

2.2 Étudiants

2.2.1 Trouver un quiz

Pour trouver un quiz, un étudiant a plusieurs possibilités. Le professeur peut donner l'url exacte du quiz à compléter, ce qui peut être pratique dans un e-mail ou toute communicaton informatisée. Une fonctionnalité permet au utilisateurs de générer un code QR correspondant à un quiz, ce qui est idéal pour afficher sur un projecteur en classe ou sur une feuille imprimée. Un étudiant peut aussi accéder à un quiz en mémorisant son id et en l'entrant dans la champ prévu à cet effet sur la page "Trouver un quiz".

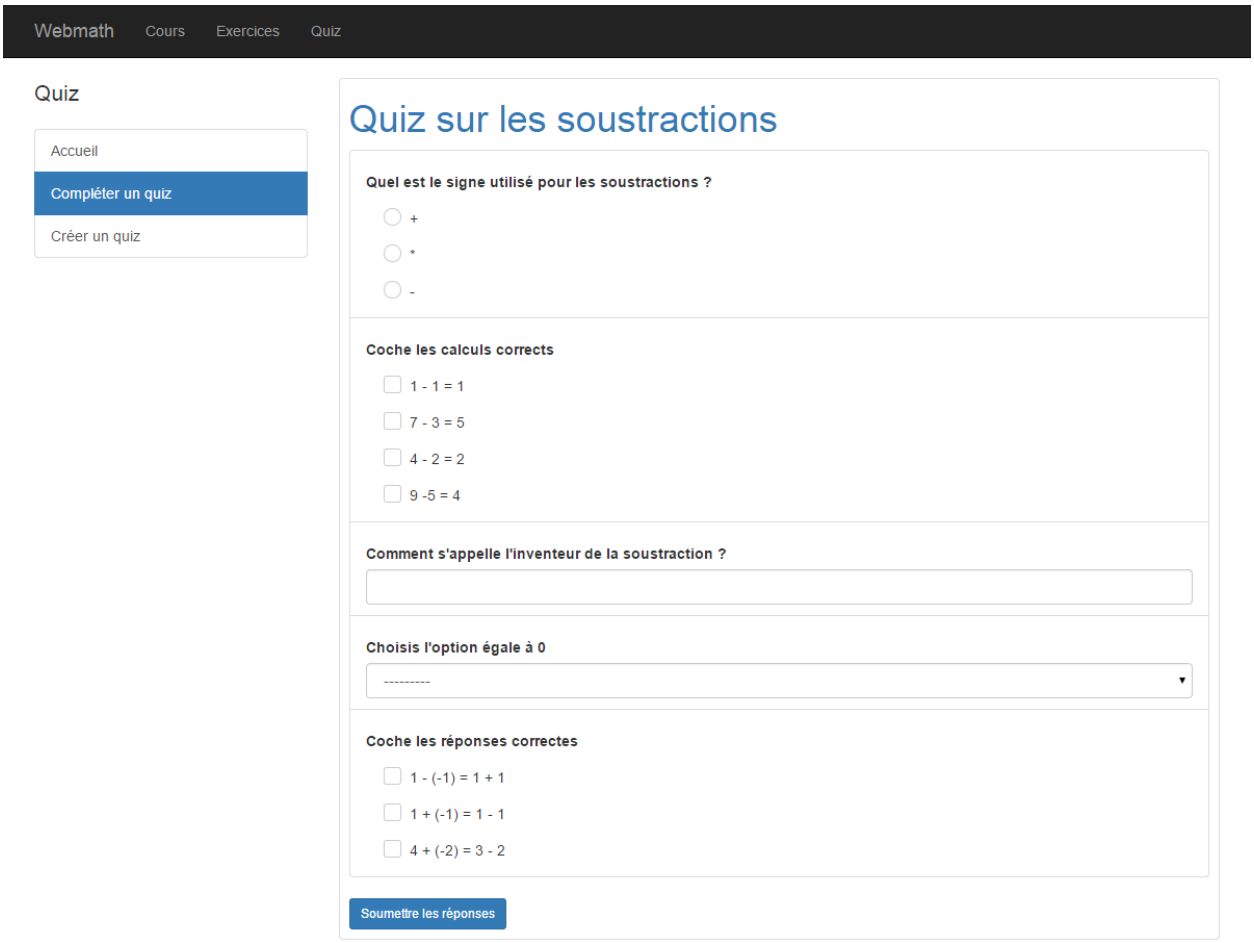


Dans son espace utilisateur, l'étudiant peut aussi consulter les derniers quizs créés dans son groupe et peut donc y accéder facilement (fonctionnalité externe à l'application).

2.2.2 Compléter un quiz et correction automatique

Une fois que l'étudiant à accédé au quiz, il peut le compléter très simplement en remplissant les champs de formulaires affichés. Lorsqu'il a fini, il peut soumettre ses réponses et il est redirigé vers une page de correction. Les réponses incorrectes sont affichées en rouge avec la solution et une explication pour chaque question. Les points reçus pour chaque question sont affichés avec le total de points sur le quiz. L'étudiant peut aussi comparer son score à la moyenne des autres étudiants du groupe. Un champ de texte est disponible pour envoyer des éventuelles remarques au professeur (signaler une erreur, poser une question). La page pour compléter un quiz ainsi que celle de la correction sont optimisées pour mobile et le design responsive s'adapte parfaitement à tous types de périphériques (ordinateur de bureau, ordinateur portable, tablette, téléphone mobile).

2.2.3 Suivi personnel


Webmath
Cours
Exercices
Quiz
Quiz
Accueil
Compléter un quiz
Créer un quiz
Quiz sur les soustractions
Quel est le signe utilisé pour les soustractions ?
+
*
-
Coche les calculs corrects
1 - 1 = 1
7 - 3 = 5
4 - 2 = 2
9 -5 = 4
Comment s'appelle l'inventeur de la soustraction ?
Choisis l'option égale à 0

Coche les réponses correctes
1 - (-1) = 1 + 1
1 + (-1) = 1 - 1
4 + (-2) = 3 - 2
Soumettre les réponses

CHAPITRE 3

Documentation du code source

3.1 views.py

quiz.views.**add_correct_answer**(*request*)

Cette fonctionnalité donne la possibilité de définir comme correcte une réponse soumise par l'étudiant à une question courte et de l'ajouter aux solutions de la question en créant une nouvelle entrée dans la table SqAnswer.

Cette vue n'est utilisée que par l'intermédiaire d'une requête Ajax et n'est accessible que si l'utilisateur est un professeur.

Paramètres de la requête HTTP POST :

—answer : Clé primaire de l'entrée dans la table SqSubmit qui doit être ajoutée aux solutions.

quiz.views.**advanced_stats**(*request*, *n_quiz*)

Récupère toutes les résolutions associés au quiz avec la clé primaire n_quiz dans la base de données et affiche des statistiques précises montrant le résultat personnel des élèves pour chaque question.

Un objet QuizForms qui créé des formulaires Django pour toutes les question du quiz est instancié pour permettre au professeur de visionner chaque question du quiz. Ce formulaire ne peut pas être envoyé et sert uniquement à l'affichage.

quiz.views.**complete**(*request*, *n_quiz*)

Cette vue est destinée à la résolution des quiz. Comme create, son comportement change selon qu'il s'agisse d'une requête HTTP de type GET ou de type POST.

Dans le cas d'une requête de type GET, la vue se contente de récupérer le quiz avec la clé primaire n_quiz et d'instancier un objet utils.submit.QuizForms à partir du quiz sélectionné. Cet objet prend en charge la création de tous les formulaires Django nécessaires pour compléter le quiz (un formulaire Django par question). Ces formulaires Django sont ensuite récupérés par l'intermédiaire de la méthode .get_forms() et placés dans le contexte de la fonction render().

La requête de type POST est utilisée pour l'enregistrement des réponses soumises au quiz par un étudiant. Là aussi, on utilise la classe QuizForms, sauf que l'instanciation se fait avec un deuxième argument, data. Cet argument contient en fait les données soumises par le biais des formulaires Django lorsque un étudiant complète le quiz. Pour s'assurer de la validité de ces données, on utilise la méthode submit.QuizForms.are_valid() de la classe QuizForms. Une fois le contrôle effectué, la méthode .save_answers() se charge d'ajouter de nouvelles entrées dans la base de donnée pour stocker les réponses soumises par l'étudiant. Pour finir, l'utilisateur est redirigé vers la page de correction correspondant aux réponses envoyées précédemment.

Paramètres de la requête HTTP POST :

—Ces paramètres dépendent du type et du nombre de questions qui constituent le quiz.

quiz.views.**completed_quizzes**(*request*)

Récupère dans la base de données toutes les entrées de la table CompletedQuiz appartenant à l'utilisateur

connecté et affiche des informations générales comme le nombre de points obtenus et un lien pour afficher la correction des réponses.

`quiz.views.`**`correct`**(*request*, *n_completed*)

Récupère dans la base de données l'entrée de la table `CompletedQuiz` avec la clé primaire `n_completed` et affiche une correction détaillée des réponses soumises par l'étudiant.

L'objet `CorrectQuiz` regroupe et calcule toutes les informations nécessaires à l'affichage du quiz. Cet objet est ensuite placé dans le contexte de la fonctions `render` et fournit ainsi un accès facilité aux données de la correction depuis le template.

`quiz.views.`**`create`**(*request*)

Cette vue sert à la fois à l'affichage de l'outil de création de quiz et à l'enregistrement de nouveaux quiz.

S'il s'agit d'une requête HTTP GET, elle renvoie simplement le template de création de quiz contenant un formulaire vide.

Au contraire, s'il la requête HTTP est de type POST, la vue se charge d'enregistrer le nouveau quiz dans la base de données en instanciant un objet de la classe *`utils.save.SaveQuiz`* avec en argument les paramètres de la requête HTTP et le compte utilisateur du créateur du quiz. Cette classe se charge ensuite de créer les entrées nécessaires dans les différentes tables de la base de données.

Une fois le quiz enregistré, la vue redirige l'utilisateur vers la page de résolution du quiz qui vient d'être créé.

Paramètres de la requête HTTP POST :

—`title` : Le titre du quiz

—`json` : Les données des questions du quiz structurées sous forme de json. Le json sera ensuite parsé par la classe `SaveQuiz`.

—`quizcode` : Le format texte du quiz, interprété côté client pour construire le json

`quiz.views.`**`created_quizzes`**(*request*)

Récupère dans la base de données toutes les entrées de la table `Quiz` créées par le professeur connecté. Les données des quiz sont utilisées pour afficher le résultat moyen des étudiants pour le quiz, la date de création ainsi qu'un lien vers la page de résolution du quiz ou vers les statistiques avancées.

`quiz.views.`**`find`**(*request*)

Récupère dans la base de données les derniers quiz créés par des professeurs et renvoie un template contenant des informations générales sur ces quiz et un lien pour y accéder.

`quiz.views.`**`findquiz`**(*request*)

Renvoie sous forme de json le titre et l'url pour accéder au quiz dont la clé primaire est donnée en paramètre de la requête GET.

L'url du quiz est construite à l'aide de la méthode `reverse(vue, arguments)` fournie par Django. Le premier argument de la fonction `reverse()` est une chaîne de caractère composée du nom de l'application Django et du nom de la vue (dans les urls) respectant le schéma suivant : `"nom_app:nom_vue"`. L'argument `args` est une liste contenant les arguments de l'URL de la page souhaitée. Ici, il s'agit simplement de la clé primaire du quiz recherché.

Cette vue est essentiellement destinée à être utilisé par une requête Ajax depuis la vue `find`.

Exemple de json renvoyé par la vue :

```
{
    "title": "Un quiz sympathique",
    "url": "/quiz/1/complete/"
}
```

Paramètres de la requête HTTP GET :

—`quiz` : Clé primaire du quiz recherché

`quiz.views.`**`getdraft`**(*request*)

Renvoie sous forme de json le titre et le code correspondant au brouillon en paramètre.

Cette vue est essentiellement destinée à être utilisé par une requête Ajax.

Exemple de données d'un brouillon en json :

```
{
    "title": "Brouillon 1",
    "code": "## Cases à cocher\n * Option 1\n= Option 4\n= Option 5"
}
```

Paramètres de la requête HTTP GET :
—`draft` : Clé primaire du brouillon

`quiz.views.`**`listdrafts`**(*request*)

Renvoie le titre et la clé primaire de tous les brouillons de l'utilisateur connecté sous forme de json.

Exemple de json renvoyé dans le cas d'un professeur ayant enregistré deux brouillons :

```
[
    {
        "title": "Brouillon 1",
        "id": 1
    },
    {
        "title": "Brouillon 2",
        "id": 2
    }
]
```

Cette vue est essentiellement destinée à être utilisé par requête Ajax.

`quiz.views.`**`savedraft`**(*request*)

Créé une nouvelle entrée dans la table `QuizDraft`. Le brouillon est associé à l'utilisateur connecté et enregistre les valeurs données en paramètres pour les colonnes `title` et `code`.

Cette vue n'est disponible que si l'utilisateur connecté appartient au groupe 'teachers'.

Paramètres de la requête HTTP POST :
—`title` : Titre du brouillon
—`code` : Code du quiz

3.2 models.py

class `quiz.models.`**`CompletedQuiz`**(**args*, ***kwargs*)

Contient

`get_questions_submits`()

Renvoie la liste des entrées des tables `SqSubmit`, `QcmSubmitOne` et `QcmSubmitMulti` correspondant à la résolution `self`. Chaque élément de cette liste représente concrètement la réponse soumise à une question du quiz.

Cette liste est triée selon l'ordre d'apparition des questions correspondant réponses proposées.

`update_total_result`()

Met à jour le nombre de points obtenus pour la résolution du quiz en entier en fonction des points obtenus pour chaque réponse soumise aux questions du quiz.

Pour comptabiliser le nombre total de points obtenus, cette méthode parcourt la liste des réponses apportées avec la méthode *`CompletedQuiz.get_questions_submits()`* appliquée à la résolution en question. calcule la somme des points obtenus pour chacune des questions.

Cette méthode peut être utilisée lorsqu'une résolution vient d'être soumise par un élève pour comptabiliser une première fois les points obtenus ou lorsqu'une entrée dans *`SqAnswer`* a été ajoutée pour mettre à jour les statistques en fonction des changements.

class `quiz.models.`**`Qcm`**(*id*, *text*, *comment*, *points*, *number*, *id_quiz_id*, *multi_answers*)

average_result()
Récupère toutes les réponses soumises pour la question et renvoie la moyenne arrondie à deux chiffres après la virgule sur la base du nombre de réponses proposées et sur la somme des résultats obtenus.

create_form(**args*, ***kwargs*)
Retourne un formulaire Django permettant à un étudiant de répondre à la question `self`. Le formulaire sera de type `RadioForm` si un seul choix peut être sélectionné et de type `CheckboxForm` si la question autorise l'étudiant à cocher plusieurs options.

Les arguments fournis lors de l'instanciation du formulaire sont analogues à ceux qui sont donnés pour l'instanciation d'un `TextForm` dans *SimpleQuestion.create_form()*.

save_submit(*data*, *completed*)
Enregistre une nouvelle entrée dans la table *QcmSubmitMulti* ou *QcmSubmitOne* selon qu'il s'agisse d'une question avec plusieurs options correctes ou une seule. Ces tables permettent de stocker le(s) choix sélectionné(s) par l'étudiant pour la question `self`.

La récupération des données du formulaire à partir de l'argument `data` et l'instanciation du modèle est analogue à la manipulation effectuée dans *SimpleQuestion.save_submit()*

class quiz.models.**QcmChoice**(*id*, *text*, *valid*, *id_question_id*)

checked(*qcmsubmit*)
Détermine si le choix a été sélectionné ou non dans la réponse soumise `qcmsubmit`.

S'il la question peut admettre plusieurs options correctes, deux conditions doivent être remplies. D'abord, au moins un choix doit avoir été coché dans `qcmsubmit`. Cette vérification est nécessaire car le champ `qcmsubmit.id_selected` peut valoir `null` dans la base de données. Ensuite, il suffit de regarder si le choix se trouve dans la liste des options sélectionnés avec la méthode `qcmsubmit.id_selected.all()`.

Dans le cas d'une question avec une seule réponse correcte, la méthode vérifie simplement que l'élément sélectionné corresponde au choix défini comme correct.

correct_submit(*qcmsubmit*)
Détermine le choix `self` a été coché correctement dans la réponse soumise `qcmsubmit`. Si l'option a été cochée et qu'elle est définie comme valide, la méthode renvoie `True`. Si l'option est cochée mais définie comme invalide, la valeur de retour sera cette fois `False`. Si l'élève n'a pas coché l'option, le raisonnement se fera de manière analogue mais dans le sens inverse.

class quiz.models.**QcmSubmitMulti**(*id*, *result*, *id_submitted_quiz_id*, *id_question_id*)

save_result()
Comptabilise et enregistre les points obtenus par rapport aux choix cochés.

Cette méthode parcourt tous les choix possibles pour la question `self.id_question` récupérés avec `Qcm.get_choices()`. Pour chaque choix, elle utilise la méthode *QcmChoice.correct_submit()* pour vérifier que le choix a été coché correctement. L'étudiant obtient une partir des points pour chaque choix correct.

class quiz.models.**QcmSubmitOne**(*id*, *result*, *id_submitted_quiz_id*, *id_question_id*, *id_selected_id*)

save_result()
Comptabilise et enregistre les points obtenus par raport au choix sélectionné.

Si le choix sélectionné correspond à la solution, tous les points sont attribués. Si aucune option n'est choisie, l'étudiant obtient automatiquement zéro point.

class quiz.models.**Quiz**(**args*, ***kwargs*)
Table dont le rôle est de stocker les informations générales sur le quiz telles que le titre du quiz, le nombre maximal de points pouvant être obtenus ou la date et l'heure de sa création.

Tous les modèles contenant les données des questions du quiz incluent une clé étrangère reliant la question avec la table `'Quiz`.

average_result()
Récupère dans la base de données toutes les résolutions se rattachant au quiz en question puis calcule la moyenne des résultats obtenus sur la base du résultat de chaque élève et le nombre de résolutions envoyées.

get_questions()
Récupère la liste de toutes les questions du quiz et les trie dans l'ordre d'apparition dans le quiz.

length()
Retourne le nombre de questions qui composent le quiz.

class quiz.models.**QuizDraft**(**args*, ***kwargs*)
La table QuizDraft sert à stocker les brouillons. Elle contient le titre du brouillon, le code ainsi qu'une relation vers le créateur du brouillon.

class quiz.models.**SimpleQuestion**(*id*, *text*, *comment*, *points*, *number*, *id_quiz_id*)

average_result()
Renvoie le nombre moyen de points obtenus pour la question en se basant sur le nombre total de réponses soumises et sur la somme de tous les points obtenus. La moyenne est ensuite arrondie à deux chiffres après la virgule pour éviter tout problème d'affichage.

Au cas où aucune réponse n'a encore été soumise, cette méthode retourne simplement la chaîne -- qui peut directement être utilisée dans le template.

create_form(**args*, ***kwargs*)
Instancie un formulaire Django personnalisé de type TextForm correspondant à la question. Ce type de formulaire est défini dans py :mod :*forms* et spécialement conçu pour les questions à réponse courte.

Lors de l'instanciation, plusieurs arguments sont fournis. Premièrement, on indique la question self pour que des informations supplémentaires la concernant puissent éventuellement être obtenues depuis le formulaire.CompletedQuiz Ensuite, l'index de la position de la question dans le quiz est attribué à l'argument prefix. Cet argument permet d'identifier la question correspondant au formulaire lorsque les données entrées par l'utilisateur sont récupérées.

get_wrong_answers()
Retourne la liste de toutes les réponses incorrectes soumises pour la question self. Pour éviter les doublons, les réponses équivalentes sont renvoyées une seule fois. Par exemple, si deux réponses valent "5", seule la première sera renvoyée par cette méthode.

Comme cette méthode est utilisée pour afficher les réponses soumises incorrectes pouvant potentiellement être définies comme correctes par la suite au cas où le professeur les juge acceptables, les réponses vides sont aussi exclues puisqu'il n'y aurait pas de sens à les admettre dans les solutions.

save_submit(*data*, *completed*)
Cette méthode permet d'enregistrer dans la base de données une réponse soumise par l'utilisateur à la question self en créant une entrée dans la table SqSubmit.

L'argument data est dictionnaire contenant les paramètres de la requête HTTP qui concernent le formulaire créé pour la question. On peut donc facilement accéder au texte entré par l'élève avec la clé 'answer'.

L'argument completed contient la référence vers l'élément de la table CompletedQuiz qui comprend les données de la résolution de l'élève. On peut ainsi facilement relier l'entrée créée dans SqSubmit avec la résolution de l'élève.

update_question_results()
Permet de réévaluer toutes les réponses soumises pour la question après l'ajout d'une solution correcte pour corriger les statistiques.

Pour ceci, la méthode récupère la liste des réponses soumises à la question dans la table SqSubmit. Elle applique la méthode .save_result à chacune d'entre elles. De plus, comme le résultat de la résolution peut changer, il faut aussi mettre à jour le résultat total obtenu pour la résolution (table CompletedQuiz) liée à chaque réponse soumise.

class quiz.models.**SqAnswer**(*id*, *text*, *id_question_id*)

class quiz.models.**SqSubmit**(*id*, *text*, *result*, *id_question_id*, *id_submitted_quiz_id*)

build_correct()
Instancie et retourne un objet CorrectSq correspondant à la réponse soumise self. La classe :py :class :CorrectSq permet un accès plus rapide aux données nécessaires à l'affichage de la correction.

correct()
Détermine si la réponse soumise est correcte en vérifiant qu'elle se se trouve dans la liste des solutions correctes.

get_corrections()
Renvoie la liste des solutions correctes pour la question sous forme de liste de chaînes de charactères

save_result()
Comptabilise et enregistre les points obtenus pour la réponse soumise. Si la réponse soumise est correcte, tous les points sont attribués. Dans le cas contraire, aucun point n'est attribué.

set_as_correct()
Si la réponse soumise à la question avait été définie comme incorrecte lors de la correction automatique, cette méthode permet d'ajouter la réponse soumise aux solutions correctes de la question.

Toutes les réponses soumises pour la question sont ensuite réeavaluées pour mettre à jour les statistiques.

3.3 forms.py

class quiz.forms.**CheckboxForm**(*queryset*, **args*, ***kwargs*)
Formulaire Django personnalisé correspondant à une question à choix multiples pouvant admettre plusieurs réponses correctes.

Le type de champ de formulaire utilisé est un ModelMultipleChoiceField. Ce champ permet de sélectionner plusieurs entrées d'une table de la base de données par l'intermédiaire de cases à cocher. Ici, il s'agit de la table :py :class'models.QcmChoice'.

L'argument queryset permet de définir les choix qui seront affichés, il s'agit d'une liste d'entrées de la table :py :class'models.QcmChoice'.

get_type()
Retourne 1 pour donner une indication sur la manière d'afficher la question dans le template

class quiz.forms.**QuestionForm**(*question*, **args*, ***kwargs*)
Classe abstraite dont héritent tous les formulaires destinés à la résolution des quiz. Seul l'attribut question est défini. Il correspond à la référence de la question dans la base de données et permet d'afficher facilement des informations sur la question depuis le template.

class quiz.forms.**RadioForm**(*queryset*, **args*, ***kwargs*)
Formulaire Django personnalisé correspondant à une question à choix multiples avec une seule réponse correcte.

Ici, on utilise un champ de formulaire ModelChoiceField. Il s'agit du même principe que pour *CheckboxForm* sauf qu'un seul choix peut être sélectionné et que le formulaire est affiché en HTML sous forme de boutons radio.

get_type()
Retourne 1 pour donner une indication sur la manière d'afficher la question dans le template

class quiz.forms.**TextForm**(**args*, ***kwargs*)
Formulaire Django personnalisé destiné à l'affichage des questions à réponse courte.

Un CharField constitue l'unique champ de ce formulaire. Il s'agit simplement d'un champ de texte basique dans lequel l'étudiant peut écrire sa réponse.

get_type()
Retourne 0 pour donner une indication sur la manière d'afficher la question dans le template

3.4 utils/submit.py

class quiz.utils.submit.**QuizForms**(*quiz*, *data=None*)

Le rôle de cette classe est de créer des formulaires Django correspondant aux questions d'un quiz en particulier puis de fournir des méthodes pour traiter les données récupérées grâce aux formulaires et de les enregistrer.

are_valid()

Applique la méthode `.is_valid()` à chaque formulaire correspondant à une question et renvoie `True` si aucun problème survient avec la validation.

save_answers(*user*)

Cette méthode permet de sauvegarder dans la base de données les réponses soumises par le biais des formulaires Django correspondant à chaque question.

Pour ce faire, une nouvelle entrée dans la table `CompletedQuiz` est d'abord créée. Ensuite, la méthode parcourt parallèlement la liste des questions du quiz et les formulaires correspondants. Pour chaque question, elle appelle la méthode `.save_submit()` avec en argument les données récupérées à partir du formulaire associé à la question. La méthode `.save_submit()` se charge ensuite de sauvegarder les données concernant les réponses soumises aux différentes questions. La manière de traiter ces données dépendera du type de question dont il s'agit. On peut appliquer la méthode `.save_submit()` à chacune des questions du quiz sans se soucier du type car son comportement est défini différement selon la classe à laquelle appartient la question.

3.5 utils/correct.py

Ensemble d'objets pour permettre un accès facilité aux données nécessaires à l'affichage des corrections

class quiz.utils.correct.**CorrectChoice**(*choice*, *qcmsubmit*)

Fournis des raccourcis pour accéder aux données des choix de QCM à corriger. Cette classe permet d'indiquer le texte à afficher avec le choix, de déterminer s'il a été coché et s'il est correct.

class quiz.utils.correct.**CorrectQcm**(*qcmsubmit*)

Facilite l'affichage de la correction des questions à choix multiples par l'intermédiaire de raccourcis et instancie des objets de type *CorrectChoice*.

L'attribut `type` vaut `1` pour les QCM à réponses multiples, `2` pour les listes déroulantes et `3` pour les QCM à réponse unique.

class quiz.utils.correct.**CorrectQuestion**(*submit*)

Définit des raccourcis pour accéder aux attributs communs à tous les types de questions.

class quiz.utils.correct.**CorrectSq**(*sqsubmit*)

Fournit des attributs spécifiques à la correction des questions à réponse courte. L'attribut `type` permet d'identifier dans le template le type de question dont il s'agit et de déterminer la manière d'afficher la correction.

Pour une question à réponse courte, `type` vaut `0`.

3.6 utils/save.py

class quiz.utils.save.**SaveQcm**(*question*, **args*, ***kwargs*)

Enregistre une question à choix multiples dans la base de données. S'il s'agit d'une question à réponses correctes multiples, il faut assigner la valeur `True` à l'attribut `multi_answers`. Par défaut, celui-ci reçoit la valeur `False`. Les différentes options à sélectionner sont également enregistrées.

add_option(*option*)

Ajoute une option à la question en créant une nouvelle entrée dans la table `models.QcmChoice`.

class quiz.utils.save.**SaveQuestion**(*question*, *Model*, *quiz_db*, *n*)

Classe abstraite qui créé une nouvelle question dans la base de données avec les attributs communs à tous les types de questions. La méthode `.save()` n'est pas encore utilisée car certains attributs doivent encore être ajoutés dans les classes filles.

La table concernée peut être `models.SimpleQuestion` ou `models.Qcm`. Cela dépend de l'argument `Model` qui définit la classe à utiliser.

L'argument `question` est un dictionnaire contenant les données sur la question. Les arguments *quiz_db*‘ et `n` correspondent respectivement à la référence du quiz dans la table :py :class :models.Quiz‘‘ et à l'index de la position de la question.

class quiz.utils.save.**SaveQuiz**(*title*, *questions_list*, *quizcode*, *teacher*)

Classe permettant l'enregistrement d'un quiz et de toutes ses questions dans la base de données. Une première entrée `models.Quiz` est enregistrée puis les données des questions sont sauvegardées par l'intermédiaire des classes *SaveSimpleQuestion* et *SaveQcm*.

class quiz.utils.save.**SaveSimpleQuestion**(*question*, **args*, ***kwargs*)

Enregistre une question à réponse courte dans la base de données. Tous les champs de la base de données ont déjà été définis par *SaveQuestion*, la méthode `.save()` peut donc être utilisée directement. Les différentes solutions sont ensuite sauvegardées.

add_answer(*text*)

Ajoute une solution à la question en créant un champ dans la table `models.SqAnswer`

CHAPITRE 4

Modèle relationnel

4.1 Introduction

Une des premières étapes importantes dans le développement d'un site web est l'élaboration d'un modèle relationnel structuré permettant de stocker toutes les données générées par l'application et de les relier entre elles. Un modèle relationnel décrit les différentes tables de la base de données et les liens entre ces tables. Une table peut être comparée à un tableau contenant des informations. Chaque table comporte une ou plusieurs colonnes, chaque colonne stockant un type de données précisément défini, par exemple un nombre entier ou une chaîne de caractères. Si on imagine une table contenant les données sur les utilisateurs d'un site, une colonne pourrait alors contenir le pseudonyme d'un utilisateur et une autre son âge. On peut ensuite ajouter des entrées dans une table, c'est à dire un ensemble de données dont chaque élément correspond à une colonne de la table. Dans l'exemple précédent, on ajouterait ainsi une ligne pour chaque utilisateur s'inscrivant sur le site. Chaque ligne est identifiée grâce à une clé primaire unique, habituellement sous la forme d'un entier, qui permet de créer des liens entre différentes lignes de différentes tables. Ces liens entre différentes tables sont appelées relations.

Voici, présenté sous forme simplifiée à l'aide d'un tableau, comment on pourrait stocker les données concernant des quiz et leurs créateurs :

Table contenant les utilisateurs :

Clé primaire	Prénom	Âge
1	Paul	26
2	Juliette	22
3	Marc	48

Table contenant les quiz :

Clé primaire	Titre du quiz	Auteur
1	Fonctions exponentielles	3
2	Logarithmes	3
3	Comportement à l'infini	2

Ici, le premier quiz a comme titre *Fonctions exponentielles*, comporte 5 questions et a été créé par Marc (Utilisateur 3).

Une fois que le modèle relationnel a été élaboré, on peut créer une base de données sous forme de fichier. Le langage SQL permet de créer les tables d'une base de données, d'y enregistrer des informations et de faire des requêtes, c'est à dire récupérer des données enregistrées. Un logiciel ou une application web peut ainsi communiquer avec une base de données et stocker les informations nécessaires de manière permanente.

Django offre la possibilité de créer une base de données et d'interagir avec celle-ci par le biais d'une ORM (Object Relational Mapper), on peut donc éviter l'utilisation du langage SQL et communiquer avec la base de données avec des objets et des méthodes Python. Cet aspect de Django sera abordé un peu plus loin.

4.2 Utilisation dans l'application de création de quiz

4.2.1 Implémentation dans Django

Comme indiqué précédemment, Django fournit une interface de haut niveau pour créer et interagir avec une base de données. On peut écrire nos tables avec une architecture orientée objet, que Django "traduira" ensuite en SQL. Ainsi, pour créer une table dans notre base de données, il suffit de définir une classe héritant de `models.Model` et d'initialiser des variables de classes pour ajouter des colonnes à la tables. Ainsi, par exemple, pour implémenter une table contenant les données générales d'un quiz, il suffit d'écrire le code suivant :

```
class Quiz(models.Model): #Infos générales sur le quiz
    title = models.CharField(max_length=100) #Colonne contenant une chaîne de caractères
    creation_date = models.DateTimeField() #Colonne contenant une date/heure
    code = models.CharField(max_length=1000) #Colonne contenant une chaîne de caractères
```

Les objets comme `CharField()` ou `DateTimeField()` permettent de définir des champs d'un type de données précis. Une liste complète des types de champs est disponible sur la documentation officielle de Django : https://docs.djangoproject.com/en/1.7/ref/models/fields/#field-types

4.2.2 Schéma global

L'élaboration d'un schéma relationnel n'est pas chose facile car il est nécessaire que celui ci réponde à certains critères. Il doit être relativement simple afin de garder une certaine flexibilité et d'avoir la possiblité d'être modifié plus tard, car il est souvent difficile de prédire à l'avance les difficultés liées au stockage des données qui pourraient être rencontrées au cours du développement. Il doit également être possible d'ajouter des fonctionnalités sans devoir revoir toute l'organisation du schéma ou créer un trop grand nombre de nouvelles tables. Malgré cela, le modèle doit aussi correspondre aux exigences des fonctionnalités de l'application et doit inclure toutes les relations nécessaires. Il s'agit donc d'une étape cruciale qui peut s'avérer décisive pour la suite du développement.

Voici le diagramme des tables utilisés pour stocker les données des quiz :

4.2.3 Explications des tables

Quiz

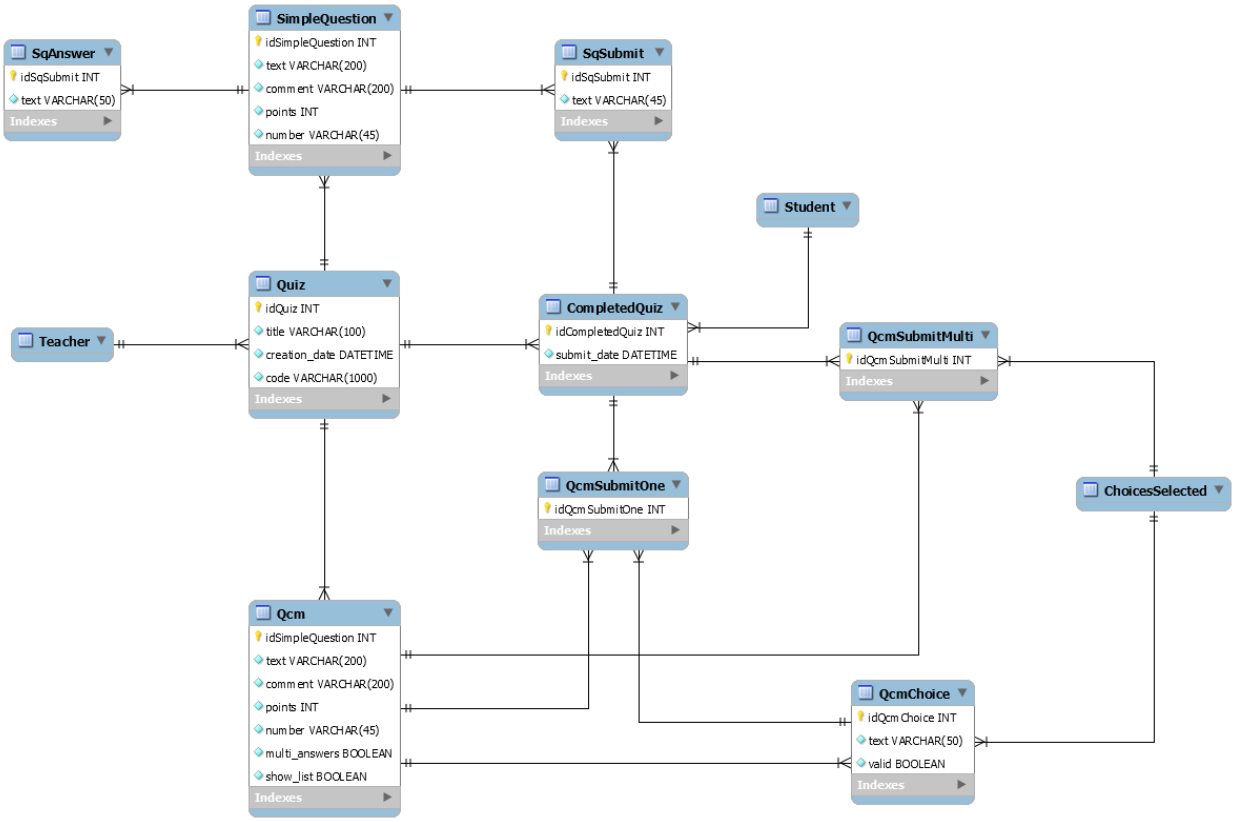
La table `Quiz` est la table centrale de l'application et toutes les autres tables s'organisent autour d'elle. Elle comporte trois colonnes importantes : une contenant le titre, une autre contenant la date et l'heure de création (ajouté automatiquement lorsqu'un nouveau quiz est créé) ainsi qu'une colonne dans laquelle est stockée le nombre maximal de points pouvant être obtenus pour le quiz en question. Cette table comporte également une relation vers une table Teacher qui permet l'intégration dans le projet de groupe.

Avec django, il est possible d'initialiser automatiquement un champ `DateTimeField` à la date/heure du moment où le modèle est instancié avec le paramètre `auto_now_add`

```
creation_date = models.DateTimeField(auto_now_add=True)
```

QuizDraft

Cette table est un peu isolée dans le schéma relationnel et n'a qu'une fonction : enregistrer le code qu'un quiz qu'un professeur n'a pas terminé et lui offrir la possibilité de le récupérer plus tard pour continuer son travail. Outre la relation vers la table `Teacher` et la colonne stockant le code du quiz, une colonne permet de stocker un titre pour pouvoir identifier rapidement un brouillon.



SimpleQuestion

Cette table contient les informations générales sur les questions simples du quiz. Ces questions sont présentées sous la forme d'un simple champ de texte lorsqu'un élève complète le quiz. Une première colonne `title` stocke l'énoncé de la question, `comment` permet d'inclure un commentaire affiché lors de la correction automatique du quiz (par exemple la démonstration d'une égalité), `points` définit le nombre de points attribués sur cette question et `number` enregistre l'ordre auquel doit apparaître la question dans le quiz. Une relation désigne le quiz qui intègre la question.

SqAnswer

Cette table contient simplement la solution de la question définie par la relation vers la table *SimpleQuestion*. Il est important de noter qu'il peut y avoir plusieurs solutions possibles pour une question et c'est la raison pour laquelle la solution n'est pas simplement stockée dans une colonne de *SimpleQuestion*.

Qcm

La table Qcm permet de stocker les informations générales à propos des questions à choix multiples. Ces questions sont affichées sous forme de boutons radio, de cases à cocher ou de liste déroulante. Cette table reprend plusieurs colonnes de la table *SimpleQuestion*. C'est pourquoi ces deux tables héritent en fait du même modèle dans django :

```
class QuizQuestion(models.Model): #Classe abstraite dont héritent toutes les questions
    text = models.CharField(max_length=200) #Énoncé
    comment = models.CharField(max_length=200, blank=True) #Commentaire
    points = models.FloatField(default=1)
    number = models.IntegerField() #Ordre de la question dans le quiz
```

```
    id_quiz = models.ForeignKey(Quiz)

    class Meta:
        abstract = True

class SimpleQuestion(QuizQuestion):
    pass #Cette table reprend simplement les mêmes colonnes que le modèle abstrait

class Qcm(QuizQuestion):
    multi_answers = models.BooleanField()
```

En plus des colonnes héritées de `QuizQuestion`, `Qcm` possède un champ de type booléen. Il s'agit de `multi_answers`, qui détermine si plusieurs options peuvent êres cochées ou non. Si ce champ vaut `True`, la question sera affiché sous forme de cases à cocher en HTML. Dans le cas contraire, elle sera affichée à l'aide de boutons radio.

QcmChoice

Cette table contient les différents choix possibles pour la question définie par la relation vers `Qcm`. Elle est formée de deux champs, le premier contenant le texte du choix et l'autre définissant par un booléen s'il est correct ou non de cocher ce choix. Une question à choix multiples doit avoir au moins deux choix possibles et au moins un choix correct. Si `multi_answers` vaut `False` dans `Qcm`, une seule option peut être correcte puisque l'étudiant n'a la possibilité de cocher qu'une seule option.

Note : D'un point de vue purement relationnel, comme il est indiqué sur le diagramme, cette table possède une relation vers une table qui sert d'intermédiaire entre *QcmChoice* et `QcmSubmitMulti`. Cette table intermédiaire crée en fait une relation de type *Complexe-Complexe*. L'implémentation de ce type de relation avec Django sera abordé plus loin.

CompletedQuiz

Comme on peut le voir sur le diagramme, à l'instar de `Quiz`, cette table occupe aussi un rôle central dans le modèle relationnel. Elle permet de faire le lien entre un quiz créé par un professeur et les réponses soumises à ce quiz par les étudiants. Elle possède donc une relation vers une table `Student`, qui définit l'étudiant ayant répondu au quiz. De l'autre côté, cette table pointe vers `Quiz` et définit logiquement le quiz auquel l'étudiant a répondu. Un seul champ est présent : la date et l'heure de la soumission des réponses.

SqSubmit

Il s'agit simplement de la réponse apportée à une question simple. La table a donc un champ `text` qui contient la réponse soumise par l'élève et un champ `result` qui stocke le nombre de points obtenus pour la question. Elle possède aussi deux relations, une vers `SimpleQuestion` pour préciser la question auquel l'élève a répondu, et une autre vers *CompletedQuiz*. La réponse soumise par l'élève sera ensuite comparée à(aux) solution(s) enregistrées pour déterminer si les points sont attribués ou non.

QcmSubmitOne et QcmSubmitMulti

Ces deux tables sont très similaires. `QcmSubmitOne` contient une relation vers l'option sélectionnée par l'étudiant dans une question à choix multiples avec *multi_answers* valant `False`, tandis que `QcmSubmitMulti` peut contenir des relations vers plusieurs options, quand `multi_answers` vaut `True`. Il s'agit donc dans le premier cas d'une relation *Complexe-Simple*, puique plusieurs lignes peuvent pointer vers la même option. Dans le deuxième cas, c'est une relation de type *Complexe-Complexe*, puisque plusieurs lignes peuvent pointer vers plusieurs options.

Dans Django, voici comment seront définies ces relations :

```
id_selected = models.ForeignKey(QcmChoice, null=True) #Relation Complexe-Simple
id_selected = models.ManyToManyField(QcmChoice, null=True) #Relation Complexe-Complexe
```

L'argument `null` vaut ici `True` car il se peut que l'étudiant ne coche aucun choix. Dans ce cas-là, il n'obtiendra dans tous les cas aucun point.

En plus de ces relations, ces tables enregistrent aussi le nombre de points obtenus par l'étudiant pour la question dans la colonne `result`.

q

T

U